UNIVERSIDADE DE CAXIAS DO SUL
CENTRO DE COMPUTAÇÃO E TECNOLOGIA DA INFORMAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FERNANDO MOLON TOIGO

O uso de uma Ontologia como Base de Conhecimento para o Desenvolvimento de um Jogo Eletrônico de Aventura

André Luis Martinotto
Orientador

Marcos Eduardo Casa
Coorientador

Caxias do Sul, Dezembro de 2014

O uso de uma Ontologia como Base de Conhecimento para o Desenvolvimento de um Jogo Eletrônico de Aventura

por

Fernando Molon Toigo

Projeto de Diplomação submetido ao curso de Bacharelado em Ciência da Computação do Centro de Computação e Tecnologia da Informação da Universidade de Caxias do Sul, como requisito obrigatório para graduação.

Projeto de Diplomação

Orientador: André Luis Martinotto
Coorientador: Marcos Eduardo Casa
Banca examinadora:

Marcos Eduardo Casa
CCTI/UCS
Elisa Boff
CCTI/UCS

"My third maxim was to endeavor always to conquer myself rather than fortune, and change my desires rather than the order of the world, and in general, accustom myself to the persuasion that, except our own thoughts, there is nothing absolutely in our power; so that when we have done our best in things external to us, all wherein we fail of success is to be held, as regards us, absolutely impossible: and this single principle seemed to me sufficient to prevent me from desiring for the future anything which I could not obtain, and thus render me contented."

René Descartes

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Jogos eletrônicos estão fazendo cada vez mais parte do cotidiano das pessoas. Mesmo as pequenas empresas desenvolvedoras de jogos até as grandes empresas costumam utilizar linguagens imperativas tais como C, C++, C# ou Java para a criação de jogos.

Entretanto, o desenvolvimento de jogos eletrônicos difere de outros tipos de softwares devido a necessidade de elementos gráficos avançados e interações em tempo real. Portanto, é necessário que o processo de desenvolvimento seja planejado de forma a diminuir futuras manutenções. Esse planejamento pode ser elaborado utilizando ontologias, que são formas de representação de conhecimento através da definição de conceitos e relacionamentos baseados em um domínio específico.

Este trabalho propõe a criação e utilização de uma ontologia para a modelagem de uma base de conhecimento para o jogo Adventure. O objetivo é separar as regras do domínio do jogo de outros elementos do desenvolvimento, como, por exemplo, a interface. Através da ontologia é possível, inclusive, tornar as regras de domínio independentes da linguagem de programação.

A ontologia criada nesse trabalho foi desenvolvida utilizando lógica de primeira ordem para descrever as regras básicas do jogo Adventure, e posteriormente, foi implementada na linguagem de programação declarativa Prolog. Essa que, por sua vez, foi integrada com uma interface gráfica que foi desenvolvida na linguagem C# para formar um protótipo do jogo Adventure.

Palavras-chave: Jogos, ontologia.

The use of an Ontology as Knowledge Base for the Development of an Eletronic Adventure Game

ABSTRACT

Eletronic games are increasingly being part of our daily life. Even for small game developers to big companies, programming languages like C, C++, C# or Java are usually used for the creation of games.

However, the development of eletronic games differs from others types of softwares due to the necessity of advanced graphical elements and real-time interactions. Therefore, it is needed that the process of development to be planned in order to decrease future maintenances. This planning can be elaborated using ontologies, in which are ways to represent knowledge through the definition of concepts and relationships based on an specific domain.

This work proposes the creation and using of an ontology for the modeling of the knowledge base of an eletronic game. The objective is to separete the domain rules of the game from others elements of the development, such as interface. Through the ontology it is possible, including, to render the domain rules independently of the programming language.

The ontology that was created in this work was developed using first-order logic to describe the basic rules of the Adventure game, and posteiorly was implemented in the declarative language Prolog. That, on the other hand, was integrated with an graphics interface that was developed with the C# language to form a playable prototype of the Adventure game.

Keywords: Games, ontology.

1 INTRODUÇÃO

Devido ao avanço das tecnologias e dos equipamentos de processamento gráfico, o mercado de jogos eletrônicos tem apresentado um crescimento significativo nos últimos anos. Além disso, os consoles de videogame estão ganhando espaço nas residências do público consumidor, tanto por estarem mais acessíveis, quanto por possuírem formas mais dinâmicas de diversão do que as que são encontradas em outras formas de entretenimento (EDERY; MOLLICK, 2008).

Tais motivações têm atraído o interesse de desenvolvedores de jogos eletrônicos. Jogos esses que costumam ser desenvolvidos a partir do uso de linguagens imperativas (paradigma procedural e orientado a objetos), tais como C, C++, C# ou Java. Entretanto, apesar dos conceitos encontrados em jogos possuírem uma definição comumente objetiva, as regras e os relacionamentos dos mesmos são mais complexos quando comparados com outras áreas de desenvolvimento. Isso faz com que sua implementação utilizando uma linguagem imperativa não seja trivial (MACHADO, 2009).

Dessa forma, outras formas de desenvolvimento tornam-se atrativas. Uma dessas alternativas é a utilização do paradigma de programação lógico, na qual a lógica do jogo não é descrita como um fluxo sequencial de comandos, mas sim através de uma definição de conceitos e regras que especificam o domínio do problema (VAREJÃO, 2004). Portanto, a utilização deste paradigma de programação torna-se atrativa para a análise e desenvolvimento de jogos eletrônicos (MACHADO, 2009).

Um gênero adequado de jogos para esse tipo de estrutura é o de aventura. De fato, em sua dissertação de mestrado, ZAFEIROPOULOS (2008) demonstra as vantagens da utilização da linguagem declarativa Prolog (*Programming in logic*) (CLOCKSIN; MELLISH, 1994), e posteriormente desenvolve um jogo com a mesma. Jogos também já foram usados de forma educativa com a intenção de incentivar os estudantes da computação na aprendizagem de linguagens declarativas, justamente pelo fato dos jogos serem visualmente mais interessantes (SILVA, 2007). Inclusive, uma *engine*[1] para integração de Prolog com XNA (MICROSOFT, 2014) foi criada de modo a tornar mais fácil a geração de elementos visuais ao fazer uso dessa linguagem (MACHADO, 2009).

Entretanto, criar sistemas com unidades genéricas, bem definidas, de baixo acoplamento e reutilizáveis sempre foi um dos grandes desafios da computação (MUSEN, 2000). Sendo que em jogos eletrônicos não é diferente. Tipicamente, os mesmos são desenvolvidos com uma linguagem em mente, o que acarreta na criação involuntária de uma dependência entre a lógica do jogo e a linguagem de programação

[1] Sistema de ferramentas para auxílio na criação e desenvolvimento de jogos eletrônicos.

(MUSEN, 2000), mesmo aquelas de paradigma declarativo.

Portanto, é desejável que as regras do jogo estejam definidas de uma forma genérica e de fácil entendimento. À vista disso, uma alternativa para a definição dessas regras consiste na utilização de Ontologias (RUSSELL et al., 2010) (MUSEN, 2000). Ontologias apresentam como propósito representar de uma forma geral o conhecimento, os conceitos e relacionamentos de um domínio específico (HOCHHALTER et al., 2005), a fim de se ter uma estrutura comum de informações. Além disso, ontologias fazem parte da engenharia do conhecimento, e não da engenharia de software (MUSEN, 2000), e podem ser descritas independentemente da linguagem que será utilizada para o desenvolvimento da base de conhecimento da aplicação.

Desta forma, esse trabalho tem como objetivo verificar o uso de uma ontologia para a representação da base de conhecimento de um jogo de aventura. O jogo escolhido foi o Adventure, que foi lançado para o Atari 2600 em 1979 (ATARI, 1980). O objetivo deste jogo consiste em encontrar um cálice encantado e transportá-lo para um castelo, evitando os dragões que rondam os mapas. As regras do jogo fazem parte da base de conhecimento desta ontologia que posteriormente foi implementada na linguagem declarativa Prolog. Já a interface foi desenvolvida na linguagem C#, sendo que a integração com a linguagem Prolog foi realizada através da *interface* de código aberto SwiPlCs (LESTA, 2014).

1.1 Objetivos

O objetivo deste trabalho consiste no desenvolvimento de um protótipo do jogo Adventure, que foi lançado em 1979 pela Atari Inc. para o console Atari 2600. Para tanto, foi utilizada uma linguagem lógica (Prolog), e as regras do universo do jogo foram fundamentadas em uma ontologia. Esta que, por sua vez, foi construída garantindo a generalidade e independência quanto as especificações da linguagem.

Para que o objetivo deste trabalho seja atendido, os seguintes objetivos específicos foram realizados:

- Construção da ontologia responsável por descrever as regras do universo do jogo Adventure;
- Desenvolvimento de um programa em Prolog equivalente a ontologia definida anteriormente. Ou seja, esse programa é responsável pelo tratamento da lógica do jogo;
- Desenvolvimento de um programa na linguagem C# responsável pela geração dos elementos gráficos, interação com o usuário e integração com o programa em Prolog.

1.2 Estrutura do Trabalho

O presente trabalho é composto de 6 capítulos, sendo que o conteúdo destes é definido de forma sintética como:

- Capítulo 1: apresenta uma introdução e os objetivos do trabalho;
- Capítulo 2: descreve o funcionamento e objetivos do jogo Adventure, bem como as regras de maior importância para a implementação;

- Capítulo 3: apresenta definições e conceitos relacionados a ontologias bem como mecanismos para a formalização de ontologias;
- Capítulo 4: é apresentada a ontologia criada a partir de lógica de primeira ordem para o protótipo do jogo Adventure;
- Capítulo 5: descreve a implementação da base de conhecimento e da aplicação;
- Capítulo 6: são apresentadas as considerações finais e trabalhos futuros;
- Apêndices A: apresenta uma listagem completa e contínua das regras criadas para a ontologia do jogo Adventure.
- Apêndices B: apresenta o diagrama de classes que contém as classes de modelo da aplicação e a imagem de mapeamento de objetos dos cenários.

2 O JOGO ADVENTURE

Jogos eletrônicos, de modo geral, podem ser divididos entre quatro gêneros: simulação, estratégia, ação e *role-playing*[1] ou aventura (APPERLEY, 2006). Simulação inclui jogos em que o jogador vivencia situações simples como a prática de algum esporte ou até ambientes mais complexos que são mais difíceis de se produzir na realidade, como a construção de cidades.

Já o gênero de estratégia pode ser subdividido em duas outras categorias: tempo real ou por turnos. Esse gênero é caracterizado pela visualização do cenário na perspectiva de um tabuleiro, onde o jogador tem uma visão macro do ambiente, comandando uma grande quantidade e variedade de indivíduos ao mesmo tempo.

Jogos de ação são definidos pela necessidade de execução de comandos específicos, na ordem correta e no tempo delimitado. Esses se assemelham à filmes desse mesmo gênero por conter uma grande quantidade de elementos de movimento, tensão, velocidade, reflexo e raciocínio rápido (ARRUDA, 2014).

Quanto aos jogos do gênero de *role-playing* ou aventura, seus objetivos baseiam-se na ideia de que um jogador deve navegar por ambientes acumulando recursos e dicas que ajudam a atingir o seu objetivo (MADEIRA, 2001).

Adventure é um jogo eletrônico de aventura, desenvolvido por Warren Robinett e lançado em 1979 pela Atari, Inc para o console Atari 2600. Esse jogo é conhecido por ser o primeiro jogo de aventura gráfico, e o primeiro a conter um *easter egg*[2]. O jogo vendeu cerca de 1 milhão de cópias, sendo o sétimo jogo mais vendido para esse console.

O objetivo do jogo Adventure é encontrar o Cálice Encantado, o qual foi roubado e escondido em algum lugar do reino por um feiticeiro perverso, e trazê-lo para o Castelo Dourado. Para isso, o jogador terá que passar por diversos cenários, incluindo castelos, labirintos e calabouços. Além disso, o jogador deverá lidar com três dragões, que são criações do feiticeiro: Yorgle, Grundle e Rhindle. Cada dragão possui características próprias, e todos farão o possível para impedir que o jogador cumpra seu propósito.

Nas próximas seções serão descritas as regras e mecânicas do jogo. Essas informações foram retiradas do manual oficial do jogo, bem como da experiência do autor sobre o jogo.

[1] Interpretação de personagem.

[2] Mensagem secreta ou piada inserida intencionalmente em mídias como programas de computador, filmes ou livros.

2.1 Itens

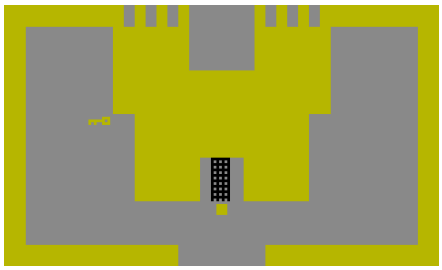
Pelo reino, o jogador pode encontrar vários itens que podem ser utilizados para auxiliar o personagem em sua aventura. Para pegar um item, basta fazer com que o personagem toque nele. Caso o personagem já esteja carregando outro item, este é solto. Itens como as chaves, por exemplo, são essenciais para a conclusão do jogo, pois sem essas o jogador não tem possibilidade de avançar nos cenários. Outros itens são opcionais e podem ser empregados para gerar atalhos, por exemplo. De forma geral, os itens existentes no jogo Adventure resumem-se às chaves, ao cálice, à espada, ao imã e à ponte.

As chaves (Figura 2.1a) são necessárias para auxiliar o personagem a percorrer os cenários. Isso devido ao fato de que para abrir o portão de um castelo de cor específica, é necessário estar com a chave de mesma cor. Ou seja, para cada castelo existe uma chave correspondente que deve ser encontrada para avançar por este castelo. Devido a sua importância, as chaves são tipicamente vigiadas por um dragão.

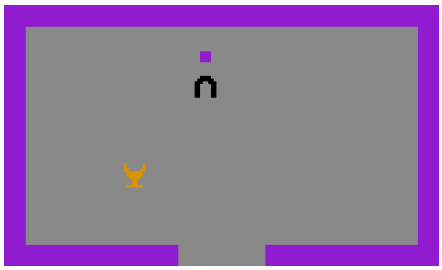
Apesar de não possuir nenhuma funcionalidade direta, o cálice (Figura 2.1b) pode ser considerado o principal item do reino pois levá-lo até o castelo dourado é a única forma de vencer o jogo.

Para derrotar um dragão é necessário neutralizá-lo. Para este fim, o jogador pode fazer uso de uma espada (Figura 2.1c). Todo dragão que encostar na espada morre, e não perseguirá mais o personagem até que o jogo seja reiniciado.

Os labirintos e calabouços podem parecer triviais, mas facilmente confundem o jogador. Para auxiliar na movimentação do personagem por esses ambientes, o jogador pode utilizar um item que funciona como uma ponte (Figura 2.1d). Isto é, se esse item for solto sobre uma parede do cenário, o personagem poderá atravessar este espaço, facilitando, assim, o avanço pelos cenários.

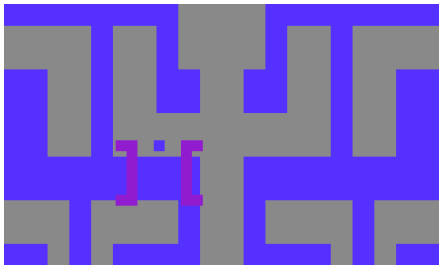


(a) Chave amarela.



(b) Cálice (em laranja) sendo atraído pelo imã (em preto).

(c) Personagem carregando a espada.



(d) Ponte (em lilás) solta em uma parede.

Figura 2.1: Os vários itens que podem ser encontrados pelos cenários.

O imã (Figura 2.1b) é outro item que pode ser empregado em certar situações. Sua principal funcionalidade é atrair outros itens para si. Esse é útil caso algum item fique preso na parede, fora do alcance do personagem.

2.2 Dragões

O maior obstáculo que o jogador encontra no Adventure são os dragões que habitam o reino. Yorgle (Figura 2.2), o dragão amarelo, é o mais lento e tem medo (ou seja, se afasta) da chave amarela. Entretanto, é o mais implacável e perseguirá o personagem até quando o mesmo trocar de cenário. Grundle, o dragão verde, é mais rápido que o primeiro, e é tipicamente encontrado vigiando o imã, a ponte ou a chave preta. Já o dragão vermelho Rhindle é o mais difícil de esquivar-se, pois é o mais veloz dentre os três. Ele pode ser encontrado vigiando a chave branca.

A forma de ataque dos dragões é a ingestão do personagem, que ocorre em duas etapas. Em uma primeira etapa, quando o dragão encosta no personagem, o mesmo pára e abre a boca, ficando em posição de morder. Em uma segunda etapa, se o personagem permanecer dentro da boca do dragão por um intervalo de tempo, ele é engolido e deve reiniciar o jogo.

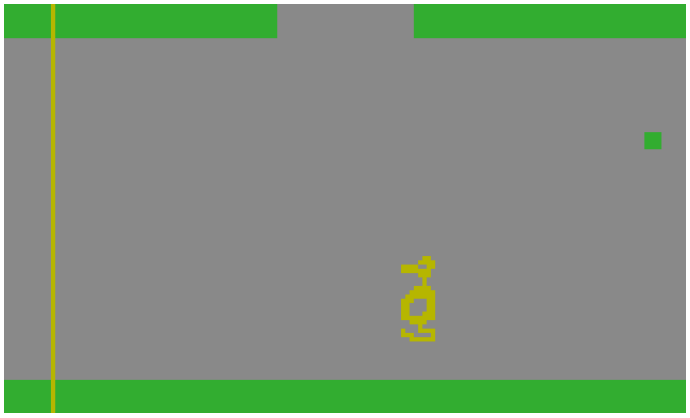


Figura 2.2: O dragão Yorgle.

2.3 Modos

O Adventure apresenta três modos de jogo diferentes. O primeiro modo consiste em uma simplificação dos outros modos, sendo adequado para a aprendizagem do funcionamento do jogo. Esse modo não possui os calabouços, tampouco o dragão Rhindle, e os itens estão disponibilizados em posições de fácil acesso.

No segundo modo, por sua vez, joga-se com o reino completo, incluindo os calabouços. Além disso, Rhindle, o dragão mais feroz, pode ser encontrado vigiando o cálice. Adicionalmente, neste modo tem-se um morcego negro que aparece frequentemente pelo cenário. O morcego carrega um item e tem como objetivo substituir o item que o personagem está carregando por esse item.

Já o terceiro modo não apresenta modificações no reino em relação ao segundo. Entretanto, todos os itens são distribuídos em posições aleatórias nos cenários. Dessa maneira, mesmo que o reino permaneça o mesmo, cada vez que o jogador iniciar o jogo ele passará por novas dificuldades e presenciará novas experiências.

Além do modo, outros parâmetros podem ser modificados de forma a alterar as mecânicas do jogo. Por exemplo, o jogador pode escolher se os dragões hesitarão por um intervalo de tempo maior que o padrão antes de finalizar a mordida (ou seja, o intervalo que ocorre entre o momento em que o dragão encostou no personagem e o momento em que o dragão o engole). De forma semelhante, pode-se configurar o jogo para que o dragão fuja quando a espada estiver no mesmo cenário em que ele se encontra.

2.4 Regras Gerais

Para efeito de simplificação, neste trabalho será desenvolvido apenas o primeiro modo do jogo, com a configuração padrão de intervalo de tempo para a mordida do dragão e os mesmos sem medo da espada ou chave. Além disso, o imã e a ponte também não serão implementados, já que esses não são necessários para o término do jogo. De forma geral, as regras do jogo nestas configurações são:

- O personagem inicia o jogo na posição de fronte ao portão do Castelo Dourado;
- O personagem pode se mover para as oito direções (cima, baixo, esquerda, direita e as 4 diagonais);
- O personagem somente pode andar na área cinza do cenário;
- O personagem não pode andar sobre os dragões, tanto vivos quanto mortos;
- O personagem pode carregar somente um item por vez;
- Ao encostar em um item, o personagem passa a segurá-lo e solta o item que está carregando, se existir;
- Um item carregado pode ser solto a qualquer momento;
- O personagem só pode entrar em um castelo se o portão do mesmo estiver aberto;
- Se uma chave de cor específica encostar no portão do castelo de mesma cor, o portão abre;
- Um dragão morre ao encostar na espada;
- O personagem ou qualquer dragão que encostar na borda do cenário passa para o cenário adjacente àquela borda;
- O dragão Grundle, enquanto vivo, persegue o personagem se estiver no mesmo cenário;
- O dragão Yorgle persegue o personagem sempre, independentemente se os mesmos se encontram em cenários distintos;
- Um dragão que encosta no personagem imediatamente toma a posição de mordida. Ou seja, o mesmo abre a boca de tal forma que o personagem fica de fronte a mesma;

- Depois de um espaço de tempo em posição de morder o dragão realiza a mordida, engolindo o personagem;
- O jogador vence quando o cálice é transportado até o Castelo Dourado;
- O jogador perde no momento que é engolido por algum dragão.

3 ONTOLOGIAS

Desde a Filosofia Antiga têm-se tentado classificar e descrever as entidades do mundo (STUDER; BENJAMINS; FENSEL, 1998). Aristóteles denominou essa área como a *metafísica*, sendo que mais tarde o filósofo Rudolf Goclenius introduziu o termo *ontologia*, que é conhecido até hoje, para representar o estudo da metafísica (MORA; TERRICABRAS, 1994).

Mais recentemente, GIARETTA; GUARINO (1995) definiram que uma ontologia que trata da natureza e da organização da realidade é chamada de Ontologia Geral ou Ontologia Formal. Eles definiram que esse tipo de ontologia concerne à descrição rigorosa de ser das coisas, por isso sua relação com a Filosofia. Na Figura 3.1 vê-se um exemplo de parte de uma ontologia universal. Em contraste, GIARETTA; GUARINO (1995) também citam que uma ontologia que busca formalizar os conceitos de um domínio específico é chamada de Ontologia Especial ou Ontologia Regional.

Atualmente, diferentes autores definem ontologias especiais de forma similar e complementar. Para STUDER; BENJAMINS; FENSEL (1998), uma ontologia consiste em uma especificação explícita e formal de um conceitualismo compartilhado. Já CORCHO; FERNÁNDEZ-LÓPEZ; GÓMEZ-PÉREZ (2003) descrevem uma ontologia como uma teoria lógica que fornece uma definição parcial e explícita para um conceito. Além disso, uma definição mais próxima à engenharia do conhecimento foi dada por NECHES et al. (1991). Para eles, uma ontologia especial é definida como os termos básicos e relacionamentos que compreendem o vocabulário de uma área, bem como as regras que combinam esses termos de forma a definir extensões desse vocabulário.

Em síntese, uma ontologia especial pode ser descrita como a definição de uma área de conceitos de maneira formal, compartilhada e explícita. Essa definição se

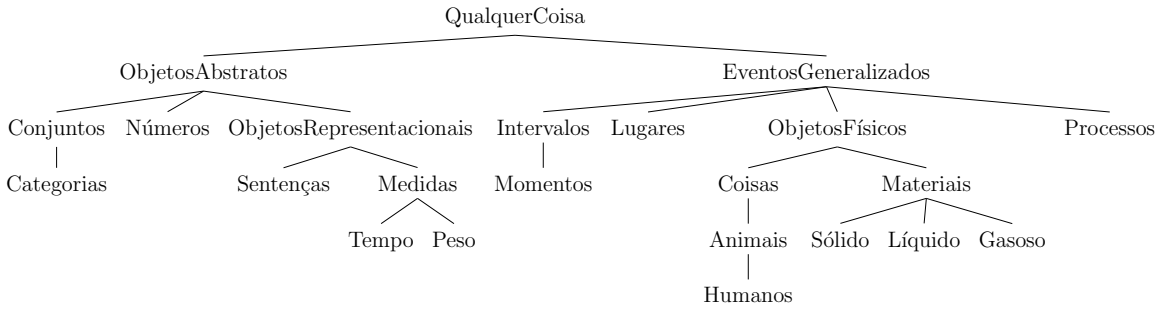


Figura 3.1: Parte de uma ontologia universal (adaptado de RUSSELL et al. (2010)).

adapta às necessidades necessárias para o desenvolvimento do jogo Adventure, uma vez que a formalidade provê uma transcrição da ontologia para o código final de uma forma de fácil interpretação, evitando o problema da ambiguidade da linguagem natural. Já o compartilhamento auxilia na compreensão dos conceitos da ontologia independentemente do desenvolvedor. Por fim, a explicitação contribui para descrever de forma clara o domínio representado pela base de conhecimento.

Na área de Inteligência Artificial, faz-se uso de ontologias especiais, diferente da definição de ontologia geral na qual visa capturar o conhecimento universal. Isso porque a mesma apresenta como finalidade resolver um problema, tal como em um sistema médico que trataria de diagnosticar uma doença a partir dos sintomas de um indivíduo (STUDER; BENJAMINS; FENSEL, 1998).

Agora, pensando no ponto de vista de um jogo, a ontologia que estamos procurando é especial (como no sistema de diagnóstico) ou universal? Dependendo do contexto, ela pode ser classificada como ambas. Quando pensamos no jogo como um domínio da nossa realidade, sua ontologia se torna especial, como uma área de conhecimento humana. Entretanto, quando analisamos como se o universo do jogo realmente existisse, verificamos que essa ontologia é geral ao ponto de vista daquele universo, ou seja, não sofre influência de outras áreas humanas que são específicas da nossa realidade. De qualquer forma, a construção da ontologia não diverge para cada tipo, então essa separação não terá influência no escopo deste trabalho. Isso somente seria necessário caso a ontologia do jogo necessitasse de uma integração com outras áreas, onde a trataríamos como uma ontologia de propósito especial.

3.1 Formalização de uma Ontologia

Para representar uma ontologia, é necessário utilizar uma linguagem para a formalização do conhecimento. Uma dessas linguagens é a lógica proposicional (ou lógica booleana), onde a ontologia é descrita através de sentenças lógicas, ou seja, sentenças que podem assumir somente os valores *verdadeiro* ou *falso* (HECK; MAY, 2013).

Na lógica proposicional tem-se as sentenças atômicas que consistem os símbolos (proposições e constantes *verdadeiro* ou *falso*) e as sentenças complexas, que são construídas a partir da união de sentenças atômicas, sendo que essa união é realizada através dos conectivos lógicos (de conjunção $\wedge$, disjunção $\vee$, condição $\rightarrow$ e bicondição $\leftrightarrow$)(FILHO; HETEM, 2012). Por exemplo, "*A chave dourada encosta no portão do castelo dourado*" e "*O portão do castelo dourado abre-se*" são sentenças atômicas e são normalmente identificadas por uma letra maiúscula. Por exemplo, essas podem ser representadas, respectivamente, pelas letras P e Q como:

P: A chave dourada encosta no portão do castelo dourado.
Q: O portão do castelo dourado abre-se.

Já a frase "*Se a chave dourada encosta no portão do castelo dourado, então o portão do castelo dourado abre-se*" é uma sentença complexa. Sua formalização na lógica proposicional pode ser definida utilizando o conector lógico de condição ($\rightarrow$) através de:

$$P \rightarrow Q$$

O uso da lógica proposicional tem a desvantagem de ser limitada quanto a descrição das regras de forma concisa. Por exemplo, podemos definir a seguinte sentença para o jogo Adventure: "*Se Yorgle encosta na espada, então Yorgle morre*". Porém, somos obrigados a repetir esta mesma regra para os outros dois dragões, pois não é possível generalizá-la. A perda da concisão está no fato de que qualquer dragão morre ao encostar na espada, e não só aquele no qual é chamado Yorgle, Grundle ou Rhindle.

Percebe-se, então, que a linguagem natural (como português ou inglês) é flexível o suficiente para descrever a regra de uma forma concisa e fácil, uma vez que podemos definir que "*Se um dragão encosta na espada, então esse dragão morre*". Todavia, linguagens naturais possuem características que não são adequadas para uma representação formal. De fato, essas dependem do contexto em que foram utilizadas, existindo a possibilidade de que sentenças causem ambiguidade, ou seja, originem dúvidas no significado da sua representação.

Para tanto, é necessário utilizar uma representação declarativa que mantenha tanto as vantagens da lógica proposicional (ou seja, usar uma semântica composicional livre de contexto e não ambígua) quanto as da linguagem natural (isto é, sua flexibilidade). À vista disso, foi criada uma extensão da lógica proposicional, chamada de lógica de primeira ordem. O domínio de modelo dessa lógica consiste de objetos (ou símbolos) e seus relacionamentos. No caso do Adventure, o domínio conteria objetos como *yorgle*, *espada*, *chave amarela*, *personagem*. Esses objetos podem se relacionar de várias maneiras, e a representação desse relacionamento é feito através de tuplas.

As tuplas são compostas por uma coleção de objetos em uma ordem específica. Por exemplo, podemos usar a tupla unária *Morto* para indicar que *yorgle* (objeto) está morto (propriedade do objeto), ou a tupla binária *Carrega* para indicar que o *personagem* (primeiro objeto) está carregando (relacionamento entre os objetos) a *espada* (segundo objeto). Tais tuplas seriam representadas através dos termos *Morto(yorgle)* e *Carrega(personagem, espada)*, respectivamente.

Com a lógica de primeira ordem introduz-se, também, os quantificadores. Esses são utilizados com o intuito de representar um grupo não especificado de objetos. Eles são dois: o universal ($\forall$) e o existencial ($\exists$). Por exemplo, foi observado anteriormente que a sentença "*Se Yorgle encosta na espada, então Yorgle morre*" não é genérica, pois é uma regra que deveria abranger qualquer dragão do jogo Adventure, e não só Yorgle. Utilizando o quantificador universal essa regra pode ser definida em lógica de primeira ordem, representando todos os objetos que são dragões. Desta forma, a sentença poderia ser escrita como:

$$\forall x \; Dragao(x) \; \wedge \; Encosta(espada, x) \rightarrow Morto(x)$$

Neste caso, cada objeto do conjunto definido pelo quantificador universal é representado pela variável x. Para a sentença, a variável x representa todos os objetos que são dragões e que ao mesmo tempo estão encostando na espada. A pronúncia comum do quantificador universal é "*Para todo...*". Dessa forma, a tradução completa da sentença ficaria "*Para todo x, se x é um dragão e x encosta na espada, então x está morto.*".

Na tupla *Encosta(espada, x)* foi definido a espada como uma constante, ou seja, um objeto específico do domínio. Isso tem a desvantagem de que, caso exista mais de uma espada no universo, a regra somente funcionará para a espada mencionada.

Para tanto, seria necessário definir que, se existe pelo menos uma espada encostada em qualquer dragão, então ele morre. O quantificador existencial apresenta essa funcionalidade. Ele é usado para estabelecer se existe, dentre o conjunto de objetos do domínio, algum (um ou mais) objeto que satisfaça a condição. Portanto, a sentença poderia ser substituída por:

$$\forall x\ Dragao(x)\ \wedge\ \exists y\ (Espada(y)\ \wedge\ Encosta(y, x)) \rightarrow Morto(x)$$

O quantificador existencial pronuncia-se "*Existe um y tal que...*". Portanto, essa sentença pode ser lida como "*Para todo x, se x é um dragão e existe um y tal que y é uma espada e x encosta em y, então x está morto.*".

Outro símbolo utilizado na lógica de primeira ordem é o da igualdade. Seu uso tem como objetivo comparar dois objetos a fim de verificar se representam uma mesma instância. O símbolo de igualdade é representado pelo símbolo =. Na sentença anterior, a função *Encosta* é utilizada para verificar se um dragão encosta em uma espada. Uma das propriedades da função *Encosta* é de que se algum objeto encosta noutro, consequentemente esse último estará encostando no primeiro. Para representar essa propriedade, podemos definir a sentença como:

$$\forall x, y\ Encosta(x, y)\ \wedge\ \neg(x = y) \rightarrow Encosta(y, x)$$

Nessa circunstância, o símbolo de igualdade com a negação ($\neg$) foi utilizado para que um objeto não possa encostar nele mesmo. Ou seja, a tupla resultado da condição só existirá quando x e y referenciarem objetos diferentes.

3.2 Construção de uma Ontologia

No processo de construção de uma ontologia é necessário seguir uma série de passos, sendo que RUSSELL et al. (2010) definem sete passos que serão descritos brevemente.

1. Identificar a tarefa: determinar que tipos de perguntas a base de conhecimento deverá responder e quais conceitos serão retornados pelas consultas.

2. Montar o conhecimento relevante: gerar, de uma maneira não necessariamente formal, o escopo da base de conhecimento e o funcionamento do domínio, evitando envolver conceitos irrelevantes a tarefa. Esse processo pode ser executado tanto pelo desenvolvedor da ontologia, quanto por algum especialista no domínio. No caso do jogo Adventure, o escopo e as regras gerais já foram definidas no Capítulo 2, através do Manual Oficial que acompanha o jogo, bem como da experiência de uso do autor.

3. Determinar um vocabulário de predicados, funções e constantes: consiste basicamente em transpor os conceitos do domínio em um formato lógico. Destaca-se que a descrição formal pode variar de acordo com o estilo do autor. Por exemplo, pode-se tratar as cores dos castelos (amarelo, vermelho, azul, etc) no reino do Adventure como tuplas (e o objeto referente àquela cor se torna um argumento) ou os objetos (castelo e chave) como tuplas, onde o argumento é a cor. O resultado dessas escolhas gerará o vocabulário da ontologia.

4. Codificar conhecimento geral sobre o domínio: gerar os axiomas correspondentes aos termos do vocabulário da ontologia. Estes axiomas podem então ser verificados pelo especialista, e, se algum erro é encontrado, deve-se retornar ao passo anterior para aplicar a devida correção.

5. Codificar uma descrição de uma instância de um problema específico: corresponde a realizar testes com entradas referentes a algum caso de uso específico do domínio. Pensando no Adventure, pode-se inserir axiomas com a localização dos principais agentes, como o personagem, os itens e os dragões para simular alguma situação específica.

6. Apresentar consultas ao sistema de inferência e obter as respostas: consiste em utilizar um sistema de inferências, realizando consultas e analisando as respostas de modo a verificar se o resultado corresponde ao esperado.

7. Depurar a base de conhecimento: analisar os axiomas de forma a identificar e solucionar inconsistências que podem ocorrer em situações específicas.

4 ONTOLOGIA DO JOGO ADVENTURE

Foram definidos, na Seção 3.2, os passos necessários para a construção da base de conhecimento de uma ontologia. Esses passos não serão seguidos rigorosamente para a construção da ontologia para o jogo Adventure, no entanto as principais partes desses poderão ser encontradas neste capítulo. Mais especificamente, o passo 1, que corresponde a identificação das consultas, será definido na Seção 4.1. Já o passo 2, que consiste em montar o escopo da base de conhecimento, foi descrito anteriormente no Capítulo 2. Na Seção 4.2 encontram-se as definições do passo 3 e o conteúdo do passo 4 é apresentado na Seção 4.3. Ou seja, as seções 4.2 e 4.3 descrevem, respectivamente, a definição das tuplas que serão utilizadas e as regras em lógica de primeira ordem que irão compor a base de conhecimento.

De fato, para a construção da ontologia, primeiramente serão identificadas as consultas que serão efetuadas pelo sistema à base de conhecimento. Isso compreende à identificação de requisitos da ontologia. Posteriormente, serão definidas as funções e as constantes que irão compor a base de conhecimento, compreendendo o processo de planejamento da ontologia. Por fim, serão definidas, em lógica de primeira ordem, as sentenças que configuram as regras do jogo Adventure.

4.1 Identificação das consultas

As regras do domínio estarão definidas em sua totalidade como tuplas na base de conhecimento, sendo que as consultas que serão efetuadas sobre a mesma serão compostas na sua maior parte por indagações sobre o estado dos agentes, tais como posição e situação atuais. O tamanho dos objetos, por serem constantes e para minimizar o número de consultas, serão obtidas somente uma vez a partir da base de conhecimento, que terá essas informações como fatos desde o início da execução. Essas constantes incluem o tamanho do personagem, das paredes do cenário, dos dragões, dos itens e dos portões.

Para que o personagem se movimente na tela é necessário que, toda vez que sua posição seja alterada, o sistema de saída tome conhecimento dessa nova posição. Portanto, a consulta "*Qual é a posição atual do personagem?*" deverá ser frequentemente executada.

Da mesma forma, os itens presentes no cenário devem movimentar-se na tela no momento em que suas posições são alteradas[1]. Então, a pergunta "*Para cada item*

[1]Os itens por si só não se movimentam pelo cenário. O ato de movimentar-se provém do movimento de um agente que carrega tal item. Como o ato de carregar é uma regra do domínio, o sistema de saída detém somente a responsabilidade de saber se o item moveu-se de lugar, independentemente do fator originário desse movimento.

contido no cenário corrente, qual é a sua posição atual?" deverá ser executada para obter essa resposta.

As informações do cenário em que o personagem se encontra também devem ser conhecidas pelo sistema de saída. Como as paredes são constantes, essas já foram identificadas na primeira consulta. Portanto, uma única consulta será necessária para a identificação do cenário em que o personagem se encontra, que é definida por "*Qual é o cenário corrente?*".

No que se refere aos dragões, uma única consulta não basta para que o sistema de saída tenha todos os dados disponíveis. Isso se deve ao fato de que um dragão possui aparências variadas de acordo com sua situação atual (normal, mordendo ou morto). Para tanto, as consultas "*Para cada dragão contido no cenário corrente, esse dragão está mordendo?*" e "*Para cada dragão contido no cenário corrente, esse dragão está morto?*" cumprirão essa tarefa. Adicionalmente, deve-se resgatar a posição atual dos dragões no cenário com a consulta "*Para cada dragão contido no cenário corrente, qual é a sua posição atual?*".

De forma similar, os portões dos castelos também apresentam formas diferentes de estado (aberto ou fechado). Portanto, além da posição atual dos portões, será necessário consultar o seu estado atual. Desta forma, as consultas "*Para cada portão contido no cenário corrente, qual é a sua posição atual?*" e "*Para cada portão contido no cenário corrente, esse portão está aberto?*" se fazem necessárias.

Por último, mas não menos importante, a consulta "*O jogador venceu o jogo?*" deve ser executada para descobrir se o jogador concluiu o jogo. Esse é exigido para que o sistema de saída saiba quando é necessário executar a animação de fim de jogo.

4.2 Definição de tuplas e nomenclaturas

O escopo da ontologia contém os conceitos e os relacionamentos incluídos no primeiro modo de jogo (vide Seção 2.3), exceto o imã e a ponte. O jogo Adventure possui somente um personagem principal que é controlado pelo jogador. Portanto, esse foi considerado como a constante *personagem*[2].

Outras constantes foram utilizadas nas regras para tratar situações específicas, como no caso do comportamento do Yorgle que é o único dragão que pode se movimentar livremente através de cenários, ou do fato de que o castelo amarelo é exclusivamente aquele que determina o fim do jogo. Esses foram representados pelas constantes *yorgle* e *casteloAmarelo*, respectivamente.

Grande parte das regras do jogo dependem de dois conceitos muito importantes: tempo e espaço. Por exemplo, para descobrir se dois objetos estão colidindo, é necessário verificar as posições de ambos no plano cartesiano. Como Adventure é um jogo em duas dimensões, foi necessário manter as coordenadas x e y de cada objeto. A tupla *Posicao*, então, foi utilizada para esse propósito, mantendo as coordenada x e y desse objeto. Quanto ao tamanho dos objetos, foi utilizada a tupla *Retangulo*, possuindo como argumentos o objeto, a altura e a largura do objeto.

Para que exista o movimento dos objetos é essencial que exista o conceito de

[2]Pode-se ao pensar que o tratamento do personagem como uma constante dificulta um desenvolvimento futuro em caso de uma possível adição de mais de um personagem no jogo. No entanto, Adventure foi criado com um jogador único em mente, e por isso essa alteração acarretaria em grandes modificações nas regras do jogo.

tempo. A movimentação de um dragão, por exemplo, se dá por um deslocamento da sua posição em relação ao tempo anterior. Para simplificar, o tempo será tratado como um inteiro crescente. Portanto, a tupla *Posicao* recebe mais um argumento que consiste no valor de tempo.

Também foram utilizadas subdivisões de objetos para facilitar o tratamento das posições nas regras. Por exemplo, os dragões tiveram sua boca e barriga definidos como diferentes objetos, cada um com sua posição e tamanho específicos, o que facilita a criação das regras de engolimento do personagem.

Quanto a nomenclatura de variáveis nas regras, serão mantidas, na medida do possível, alguns padrões. As variáveis de coordenadas x e y serão utilizadas quando forem únicas na regra. Caso não sejam únicas, e se essas coordenadas representarem a posição de algum objeto que é definido na própria regra, introduz-se um prefixo nessas coordenadas de acordo com o objeto representado. Por exemplo, a posição do personagem na regra poderá ser definido como *px* e *py*. Se essas múltiplas coordenadas representarem a posição de objetos genéricos, essas podem ser identificadas com a adição de um sufixo de valor crescente (*x1*, *x2*, *x3*, etc).

Já os objetos em si serão identificados pela primeira letra de seu tipo: dragões serão *d*, itens *i*, etc. Em caso de conflitos, como na identificação de transição e tempo na mesma regra, o uso de *t* como representação do tempo tem prioridade sobre a transição, que por conseguinte será identificada por *tr*.

A listagem de todas as tuplas definidas na ontologia, bem como a quantidade de argumento e um breve descritivo são apresentados nas Tabelas 4.1, 4.2 e 4.3. A Tabela 4.1 lista as tuplas que não dependem de tempo. Já na Tabela 4.2 encontram-se as tuplas que são dependentes do tempo. A Tabela 4.3, por sua vez, lista as tuplas que representam ações (essas são conjugadas no infinitivo).

Na Figura 4.1, vê-se os objetos da base de conhecimento em forma de árvore, de tal forma que os nodos filho são especializações do nodo pai. Como a ontologia do jogo Adventure possui uma amostra pequena de regras, que são bastante específicas e dispensam especialização, não há uma representação em árvore de todas.

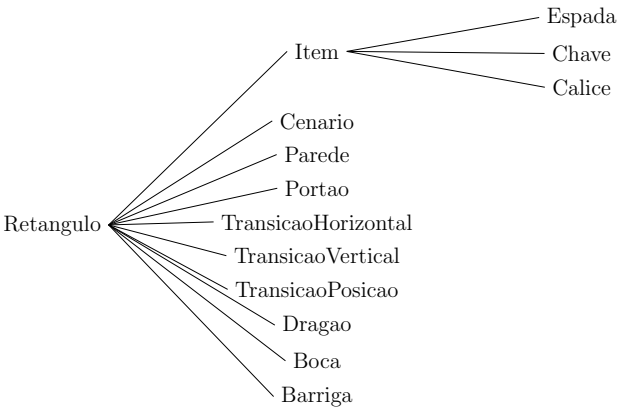


Figura 4.1: Ontologia dos objetos do protótipo do jogo Adventure.

Tabela 4.1: Tuplas que não dependem do tempo.

Nome	**Ordem**	**Argumentos**	**Descrição**
Cenario	2	cen, cor	Objeto *cen* é um cenário de cor *cor*.
Dragao	1	d	Objeto *d* é um dragão.
Boca	2	bc, d	Objeto *bc* é a boca do dragão *d*.
Barriga	2	br, d	Objeto *br* é a barriga do dragão *d*.
Item	1	i	Objeto *i* é um item.
Espada	1	e	Objeto *e* é uma espada.
Chave	2	ch, cor	Objeto *ch* é uma chave de cor *cor*.
Calice	1	ca	Objeto *ca* é um cálice.
Portao	2	po, cor	Objeto *po* é um portão de cor *cor*.
TransicaoHorizontal	4	tr, tpx, tpy, cen	Objeto *tr* é uma transição horizontal para a posição [*tpx*,*tpy*] do cenário *cen*.
TransicaoVertical	4	tr, tpx, tpy, cen	Objeto *tr* é uma transição vertical para a posição [*tpx*,*tpy*] do cenário *cen*.
TransicaoPosicao	4	tr, tpx, tpy, cen	Objeto *tr* é uma transição para a posição [*tpx*,*tpy*] do cenário *cen*.
Retangulo	3	o, lar, alt	Objeto *o* é um retângulo de largura *lar* e altura *alt*.
Parede	2	par, cen	Objeto *par* é uma parede do cenário *cen*.
Colide	8	x1, y1, lar1, alt1, x2, y2, lar2, alt2	O retângulo de posição [*x1*,*x1*], largura *lar1* e altura *alt1* colide com o retângulo de posição [*x2*,*x2*], largura *lar2* e altura *alt2*.
Contem	8	x1, y1, lar1, alt1, x2, y2, lar2, alt2	O retângulo de posição [*x1*,*x1*], largura *lar1* e altura *alt1* contém totalmente o retângulo de posição [*x2*,*x2*], largura *lar2* e altura *alt2*.

Tabela 4.2: Tuplas que dependem do tempo.

Nome	**Ordem**	**Argumentos**	**Descrição**
Encosta	3	o1, o2, t	Objeto *o1* encosta no objeto *o2* no tempo *t*.
EncostaTeste	3	o1, o2, t	Objeto *o1* encosta no objeto *o2* usando posições de teste no tempo *t*.
ContemObjeto	3	o1, o2, t	Objeto *o1* contém o objeto *o2* no tempo *t*.
ContemObjetoTeste	3	o1, o2, t	Objeto *o1* contém o objeto *o2* usando posições de teste no tempo *t*.
Carregando	2	o, t	Objeto *o* está sendo carregado pelo personagem em *t*.
CenarioCorrente	2	cen, t	Objeto *cen* é o cenário corrente no tempo *t*.
Direcao	5	o1, o2, dx, dy, t	A direção do objeto *o1* até o objeto *o2* é de [*dx*,*dy*] no tempo *t*.
Morto	2	d, t	Dragão *d* está morto no tempo *t*.
Aberto	2	o, t	Objeto *o* está aberto no tempo *t*.
Mordendo	3	d, t1, t2	Dragão *d* está mordendo entre o tempo *t1* e o *t2*.
Posicao	4	x, y, o, t	Objeto *o* está na posição [*x*,*y*] no tempo *t*.
PosicaoTeste	4	x, y, o, t	Objeto *o* está na posição de teste [*x*,*y*] no tempo *t*.
PodeMovimentar	2	o, t	Objeto *o* pode se movimentar para a posição de teste no tempo *t*.
FimJogo	1	t	O jogo terminou em *t*.

Tabela 4.3: Tuplas que representam ações.

Nome	Ordem	Argumentos	Descrição
Movimentar	4	dx, dy, o, t	Objeto *o* possui uma ação de movimento com direção [*dx*,*dy*] no tempo *t*.
Transicionar	4	tpx, tpy, o, t	Objeto *o* possui uma ação de transição para a posição [*tpx*,*tpy*] no tempo *t*.
Engolir	3	d, o, t	Objeto *d* possui uma ação de engolir o objeto *o* no tempo *t*.
TrocarCenarioCorrente	2	cen, t	Existe uma ação de troca do cenário corrente para o cenário *cen* no tempo *t*.
Largar	1	t	Existe uma ação de largar o item que o personagem está carregando no tempo *t*.
Carregar	2	i, t	Existe uma ação de carregar o item *i* pelo personagem em *t*.
Morder	2	d, t	Dragão *d* possui uma ação de morder no tempo *t*.

4.3 Definição das regras

Nesta seção, serão apresentadas as regras formalizadas em lógica de primeira ordem. Mais especificamente, na Seção 4.3.1, serão definidas as regras que não são específicas para um tipo específico de objeto. Já nas seções posteriores, serão descritas as regras mais características dos objetos, entre elas: as da movimentação do personagem, dos itens, das transições, dos dragões, da morte do personagem, dos portões e do fim de jogo.

4.3.1 Regras gerais

Cada item do jogo possui sua própria funcionalidade: a chave abre portões, a espada mata dragões etc. As funcionalidades de cada item são tratadas em regras específicas. Entretanto, alguns comportamentos são compartilhados por todos os itens, ou seja, independem da sua funcionalidade. Por exemplo, todo item é carregado quando o jogador o encosta. Desta forma, torna-se necessária a utilização de uma regra que defina de uma forma genérica quais são os itens do jogo. Ou seja, a regra abaixo define que qualquer chave, espada ou cálice pode ser considerado como um item, e, portanto, compartilha de suas funcionalidades.

$$\forall i \ (\exists cor \ Chave(i, cor)) \ \vee \ Espada(i) \ \vee \ Calice(i) \ \rightarrow Item(i)$$

Grande parte das regras necessitam verificar se dois objetos estão colidindo, ou seja, se estão se encostando. Essas regras utilizarão as tuplas *Colide* e *Encosta*. Uma vez que todos objetos que farão parte do ambiente do jogo serão representados por

retângulos, se faz necessária a criação de uma função que verifique se dois retângulos estão colidindo (tupla *Colide*) entre si. Essa pode ser definida pela sentença:

$$\forall x1, y1, lar1, alt1, x2, y2, lar2, alt2\ (\neg(x1 > x2 + lar2 - 1\ \vee\ x1 + lar1 - 1 < x2\ \vee \\ y1 > y2 + alt2 - 1\ \vee\ y1 + alt1 - 1 < y2)) \\ \rightarrow Colide(x1, y1, lar1, alt1, x2, y2, lar2, alt2)$$

Já a tupla *Encosta* utiliza a tupla *Colide* e tem como objetivo verificar se dois objetos estão colidindo. Essa tupla recebe como argumentos os objetos e o tempo de validação e faz uso das tuplas *Retangulo* e *Posicao* para buscar os dados sobre os objetos. O comportamento da função *Encosta* é definido pela sentença:

$$\forall o1, x1, y1, lar1, alt1,\ (Retangulo(o1, lar1, alt1)\ \wedge\ Retangulo(o2, lar2, alt2)\ \wedge \\ o2, x2, y2, lar2, alt2, t\ Posicao(x1, y1, o1, t)\ \wedge\ Posicao(x2, y2, o2, t)\ \wedge \\ Colide(x1, y1, lar1, alt1, x2, y2, lar2, alt2)) \\ \rightarrow Encosta(o1, o2, t)$$

Em algumas regras, como na movimentação do personagem, é necessário fazer um teste de colisão considerando a posição deslocada do personagem. Para isso, faz-se uso da regra *EncostaTeste*, que tem a mesma estrutura da regra *Encosta*, mas que utiliza a posição deslocada do objeto, e não a posição atual:

$$\forall o1, x1, y1, lar1, alt1,\ (Retangulo(o1, lar1, alt1)\ \wedge\ Retangulo(o2, lar2, alt2)\ \wedge \\ o2, x2, y2, lar2, alt2, t\ PosicaoTeste(x1, y1, o1, t)\ \wedge\ PosicaoTeste(x2, y2, o2, t)\ \wedge \\ Colide(x1, y1, lar1, alt1, x2, y2, lar2, alt2)) \\ \rightarrow EncostaTeste(o1, o2, t)$$

O estabelecimento da tupla de *PosicaoTeste* é feita a partir da ação de *Movimentar*. Ou seja, quando existe uma ação de movimento, existe uma posição final de teste para o objeto em movimento. Para os objetos que não possuem ação de movimentar, a posição final de teste se mantém inalterada em relação à posição inicial. As duas regras são representadas, respectivamente, como:

$$\forall o, t\ (\exists dx, dy\ Movimentar(dx, dy, o, t)\ \wedge \\ \exists x, y\ Posicao(x, y, o, t)) \\ \rightarrow PosicaoTeste(x + dx, y + dy, o, t)$$

e

$$\forall o, t\ (\neg\exists dx, dy\ Movimentar(dx, dy, o, t)\ \wedge \\ \exists x, y\ Posicao(x, y, o, t)) \\ \rightarrow PosicaoTeste(x, y, o, t)$$

Em outros casos, se faz necessário verificar se um retângulo está totalmente contido noutro, como é o caso da regra que verifica se o cálice está contido no castelo amarelo. Para tanto, a tupla *Contem* foi criada para descrever esse comportamento:

$$\forall x1, y1, lar1, alt1, x2, y2, lar2, alt2\ (x1 <= x2\ \wedge\ x1 + lar1 - 1 >= x2 + Lar2 - 1\ \wedge \\ y1 <= y2\ \wedge\ y1 + alt1 - 1 >= y2 + Alt2 - 1) \\ \rightarrow Contem(x1, y1, lar1, alt1, x2, y2, lar2, alt2)$$

Do mesmo modo que a colisão, foi criada uma tupla de simplificação para verificar se um objeto está totalmente contido noutro. Essa tupla é chamada de *ContemObjeto*, e os argumentos são os mesmos da tupla *Encosta*:

$$\begin{aligned}\forall o1, x1, y1, lar1, alt1,\ (Retangulo(o1, lar1, alt1)\ \wedge\ Retangulo(o2, lar2, alt2)\ \wedge \\ o2, x2, y2, lar2, alt2, t\ Posicao(x1, y1, o1, t)\ \wedge\ Posicao(x2, y2, o2, t)\ \wedge \\ Contem(x1, y1, lar1, alt1, x2, y2, lar2, alt2)) \\ \rightarrow ContemObjeto(o1, o2, t)\end{aligned}$$

Além disso, a tupla *ContemObjetoTeste* foi utilizada para fazer o mesmo tratamento da regra anterior, mas considerando a posição futura dos objetos. Essa regra é definida por:

$$\begin{aligned}\forall o1, x1, y1, lar1, alt1,\ (Retangulo(o1, lar1, alt1)\ \wedge\ Retangulo(o2, lar2, alt2)\ \wedge \\ o2, x2, y2, lar2, alt2, t\ PosicaoTeste(x1, y1, o1, t)\ \wedge\ PosicaoTeste(x2, y2, o2, t)\ \wedge \\ Contem(x1, y1, lar1, alt1, x2, y2, lar2, alt2)) \\ \rightarrow ContemObjetoTeste(o1, o2, t)\end{aligned}$$

4.3.2 Movimentação do personagem

O movimento do personagem se dará por ações. Quando o jogador pressionar uma das quatro teclas de movimento, o sistema de entrada acrescentará um fato de movimento (fato *Movimentar*) à base de conhecimento. Essa ação conterá um vetor com a direção de deslocamento, o objeto alvo do movimento e o tempo como argumentos. Por exemplo, um movimento para a diagonal superior direita conteria um vetor com coordenada x de valor 1 e coordenada y de valor 1. Já um movimento para a diagonal superior esquerda conteria um vetor de coordenadas x e y com valores -1 e 1, respectivamente.

Entretanto, a ocorrência de uma ação de movimento não corresponde diretamente à alteração de posição do personagem. De fato, a ação de movimento somente alterará a posição do personagem se a mesma for possível dado o contexto atual dos objetos. Por exemplo, o movimento não pode ocorrer caso acarrete em uma colisão com a parede. Assim sendo, a tupla *PodeMovimentar* que recebe o objeto e o tempo como argumentos será utilizada para validar a possibilidade de execução da ação. A regra de movimentação do personagem, portanto, é definida pela sentença:

$$\begin{aligned}\forall t\ (\exists x, y\ Posicao(x, y, personagem, t)\ \wedge \\ \exists dx, dy\ Movimentar(dx, dy, personagem, t)\ \wedge \\ PodeMovimentar(personagem, t)\ \wedge \\ \neg FimJogo(t)\ \wedge \\ \neg(\exists tx, ty\ Transicionar(tx, ty, personagem, t))\ \wedge \\ \neg(\exists d\ Dragao(d)\ \wedge\ Engolir(d, personagem, t))) \\ \rightarrow Posicao(x + dx, y + dy, personagem, t + 1)\end{aligned}$$

Percebe-se que na regra também é feita a verificação de existência de uma ação de *Transicionar* ou de *Engolir* no tempo t. Isso se deve pois essas ações, como a ação *Movimentar*, alteram a posição do personagem. Então, para evitar conflitos, foi decidida uma ordem de prioridades para tais ações, sendo que essa ordem consiste

em, primeiramente, a ação de *Engolir*, seguida pela ação de *Transicionar* e por último a de *Movimentar*. Portanto, a ação de movimento somente ocorrerá quando nenhuma das outras ações existirem naquele tempo.

A regra *PodeMovimentar* deve realizar a verificação de colisão entre qualquer objeto que possa bloquear a passagem do personagem. Portanto, ela deverá tratar da colisão com as paredes do cenário corrente, com os dragões (exceto se o personagem estiver na boca do dragão enquanto o mesmo estiver *mordendo*) e com os portões fechados. Como a tupla *Encosta* faz a verificação através da posição corrente dos objetos, essa não testaria a posição deslocada, ou seja, a posição final do personagem após o movimento. Por isso, foi utilizada a função *EncostaTeste*. Desta forma, a sentença de *PodeMovimentar* é definida por:

$$\begin{aligned}\forall o,t\ \neg(\exists c, par\ CenarioCorrente(c,t)\ \wedge\ Parede(par,c)\ \wedge\ EncostaTeste(par,o,t))\ \wedge\\ (\forall d\ Dragao(d)\ \wedge\ \neg EncostaTeste(d,o,t)\ \vee\\ (\neg Morto(d,t)\ \wedge\ Mordendo(d,t1,t2)\ \wedge\ \exists bc\ Boca(bc,d)\ \wedge\\ (ContemObjetoTeste(bc,o,t)\ \vee\\ (EncostaTeste(bc,o,t)\ \wedge\ \neg ContemObjetoTeste(d,o,t)))))\\ \neg(\exists po, cor\ Portao(po,cor)\ \wedge\ \neg Aberto(po,t)\ \wedge\ EncostaTeste(po,o,t))\\ \rightarrow PodeMovimentar(o,t)\end{aligned}$$

E quanto às instâncias de tempo em que não houve ação de movimento, qual será a posição do personagem no tempo seguinte? Dentro desse contexto, torna-se necessário definir uma regra para a posição do personagem em uma passagem de tempo que não houve movimentação, ou seja, quando a posição desse manteve-se inalterada. A sentença

$$\begin{aligned}\forall t\ (\exists x,y\ Posicao(x,y,personagem,t)\ \wedge\\ (FimJogo(t)\ \vee\\ \neg(\exists dx,dy\ Movimentar(dx,dy,personagem,t))\ \vee\\ \neg PodeMovimentar(personagem,t))\ \wedge\\ \neg(\exists tx,ty\ Transicionar(tx,ty,personagem,t))\ \wedge\\ \neg(\exists d\ Dragao(d)\ \wedge\ Engolir(d,personagem,t)))\\ \rightarrow Posicao(x,y,personagem,t+1)\end{aligned}$$

define essa regra. Ou seja, se existe um tempo t em que o personagem está na posição $[x,y]$ e não existe uma ação de troca de posição do personagem no tempo t, então a posição do personagem em *t1+1* será a mesma de t.

4.3.3 Carregamento de itens

O ato de carregar um item tem início no encontro do personagem com o item. Ou seja, quando o personagem encostar no item, esse passará a ser carregado pelo personagem. Assim, a regra

$$\forall t\ \exists i\ Item(i)\ \wedge\ Encosta(personagem,i,t) \rightarrow Carregar(i,t)$$

define o ato de carregar. Da mesma forma que o movimento, o ato de carregar não faz com que o personagem realmente carregue o item, somente implica que em um

tempo específico houve uma ação de carregar. O fato de que o personagem está efetivamente carregando um item é definido pelas sentenças:

$$\forall t\ \exists i\ Item(i)\ \wedge\ Carregar(i,t) \rightarrow Carregando(i,t+1)$$

e

$$\begin{aligned}\forall t\ (\exists i\ Item(i)\ \wedge\ Carregando(i,t)\ \wedge\ \neg Largar(t)\ \wedge \\ \neg(\exists i2\ Item(i2)\ \wedge\ Carregar(i2,t)\ \wedge\ \neg(i=i2))) \\ \rightarrow Carregando(i,t+1)\end{aligned}$$

A primeira regra define que um item está sendo carregado após existir uma ação de *Carregar*. A segunda concerne a permanência de carregamento do item. Em outras palavras, o item continuará sendo carregado nos tempos subsequentes até que não ocorra uma ação de *Largar* inserida pelo sistema de entrada ou enquanto o personagem não encostar em outro item (o que geraria uma nova ação de *Carregar*).

Além disso, enquanto um item está sendo carregado, sua posição precisa ser atualizada conforme o personagem se movimenta. A regra

$$\begin{aligned}\forall t\ (\exists i\ Item(i)\ \wedge\ Carregando(i,t)\ \wedge \\ \exists x,y,lar,alt\ Posicao(x,y,personagem,t)\ \wedge\ Retangulo(i,lar,alt)) \\ \rightarrow Posicao(x,y-alt-12,i,t)\end{aligned}$$

define que a posição de um item sendo carregado seja a mesma que a posição do personagem, somado com a altura do item e com um deslocamento fixo (neste caso foi definido com valor 12 (na interface representará um valor em pixels) na regra, mas esse pode variar de acordo com a implementação). Esse deslocamento é necessário para que o personagem e o item não fiquem sobrepostos, o que geraria ações de *Carregar* a todo momento.

Para os itens que não estão sendo carregados, a posição desses no tempo seguinte permanece a mesma, e é definida pela regra:

$$\begin{aligned}\forall t\ \exists i\ Item(i)\ \wedge\ \neg Carregando(i,t)\ \wedge\ Posicao(x,y,i,t) \\ \rightarrow Posicao(x,y,i,t)\end{aligned}$$

4.3.4 Transições de cenário

O cenário corrente é alterado no momento em que o personagem encosta na borda dos cenários ou na entrada dos castelos. Essas trocas de cenário, chamadas de transições de agora em diante, além de alterar o cenário corrente, também influenciam na posição do personagem. Portanto, foram definidos, juntamente com cada cenário, seus pontos de transição, representados por retângulos. Quando o personagem encostar em um desses retângulos de transição, o mesmo será portado para a posição de destino correspondente àquela transição. A Figura 4.2 demonstra um exemplo desse mecanismo, onde as transições do cenário *C1* são representadas pelos retângulos vermelhos, e as do cenário *C2* são representadas pelos retângulos azuis.

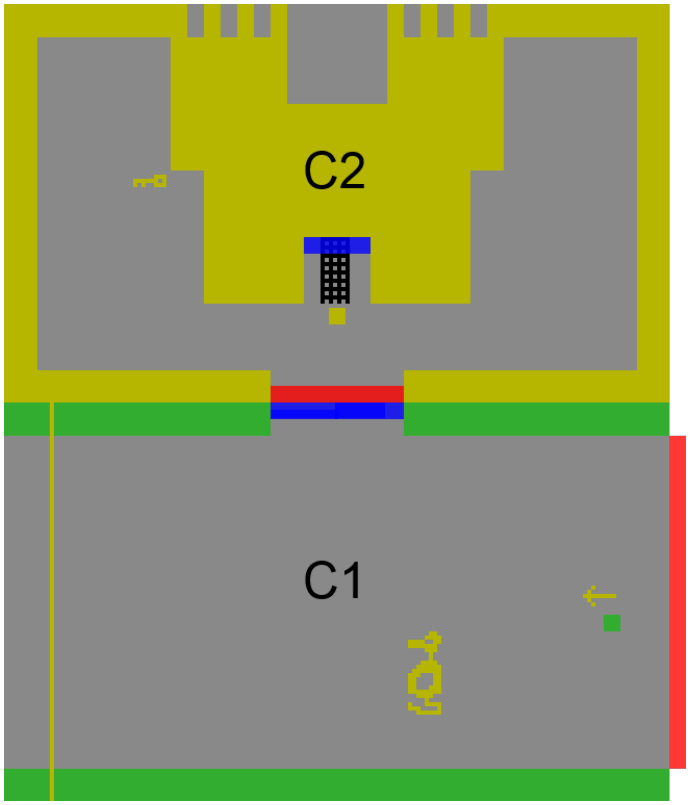


Figura 4.2: Exemplo de transições entre cenários. Os retângulos vermelhos representam as transições do cenário *C1*, e os azuis do cenário *C2*.

Dessa forma, pode-se separar as transições em três tipos: horizontal, vertical e de posição. Nas transições horizontais (onde o personagem se desloca para a borda direita ou esquerda do cenário), a posição de destino do personagem é composta pela coordenada x da posição de destino da transição e a coordenada y que é constituída da posição de destino somada com um deslocamento. Esse deslocamento é a diferença em y entre a localização atual do personagem e a própria localização da transição, e tem por objetivo manter a mesma orientação vertical na troca de mapas. Por exemplo, se o personagem encostar no retângulo de transição em uma posição próxima ao topo desse retângulo, ele deverá sair na posição de destino com a mesma diferença entre ele e o topo. A regra que demonstra esse comportamento é definida por:

$$\begin{aligned}
&\forall t\ (\exists tr, tpx, tpy, cen\ TransicaoHorizontal(tr, tpx, tpy, cen) \wedge \\
&\quad \neg CenarioCorrente(cen, t) \wedge \\
&\quad Encosta(tr, personagem, t)) \wedge \\
&\quad \exists tx, ty\ Posicao(tx, ty, tr, t)\ \wedge\ \exists px, py\ Posicao(px, py, personagem, t) \\
&\qquad \rightarrow Transicionar(tpx, tpy + py - ty, personagem, t) \wedge \\
&\qquad\quad TrocarCenarioCorrente(cen, t)
\end{aligned}$$

Contrariamente, a posição final do personagem quando o mesmo encosta em uma transição vertical (onde o personagem se desloca para a borda superior ou inferior do cenário) é composta pela coordenada x da posição de destino da transição somada com o deslocamento horizontal entre o retângulo e o personagem. Por exemplo, se

o personagem encostar no retângulo de transição em uma posição mais próxima a esquerda desse retângulo, ele deverá sair com esse mesmo deslocamento na posição de destino. O valor da coordenada y final é a posição de destino da transação. A regra correspondente é definida por:

$$\begin{aligned}\forall t\ (&\exists tr, tpx, tpy, cen\ TransicaoVertical(tr, tpx, tpy, cen) \wedge \\ &\neg CenarioCorrente(cen, t) \wedge \\ &Encosta(tr, personagem, t)) \wedge \\ &\exists tx, ty\ Posicao(tx, ty, tr, t)\ \wedge\ \exists px, py\ Posicao(px, py, personagem, t) \\ &\quad \rightarrow Transicionar(tpx + px - tx, tpy, personagem, t) \wedge \\ &\quad\ TrocarCenarioCorrente(cen, t)\end{aligned}$$

Já as transições de posição atuam de forma que o personagem seja deslocado diretamente para a posição de destino da transação. Por exemplo, se o personagem encostar na transição do portão do castelo, o mesmo deve ser portado para a entrada do cenário que corresponde a parte interna do castelo. Esse procedimento é determinado pela sentença:

$$\begin{aligned}\forall t\ (&\exists tr, tpx, tpy, cen\ TransicaoPosicao(tr, tpx, tpy, cen) \wedge \\ &\neg CenarioCorrente(cen, t)) \wedge \\ &Encosta(tr, personagem, t) \\ &\quad \rightarrow Transicionar(tpx, tpy, personagem, t) \wedge \\ &\quad\ TrocarCenarioCorrente(cen, t)\end{aligned}$$

A efetivação da ação de *Transicionar*, que é resultado das regras de transição horizontal, vertical e de posição é definida por:

$$\begin{aligned}\forall t\ (&\exists tpx, tpy\ Transicionar(tpx, tpy, personagem, t) \wedge \\ &\neg(\exists d\ Dragao(d)\ \wedge\ Engolir(d, personagem, t))) \\ &\quad \rightarrow Posicao(tpx, tpy, personagem, t + 1)\end{aligned}$$

Além disso, a tupla *TrocarCenarioCorrente* desempenha a função de troca de cenário. Portanto, é necessário fazer o tratamento dessa ação para efetivamente trocar o cenário. Esse tratamento é definido pelas sentenças:

$$\forall t\ \exists cen\ TrocarCenarioCorrente(cen, t) \rightarrow CenarioCorrente(cen, t + 1)$$

e

$$\begin{aligned}\forall t\ (&\exists cen\ CenarioCorrente(cen, t) \wedge \\ &\neg(\exists cen2\ TrocarCenarioCorrente(cen2, t)\ \wedge\ \neg(cen == cen2))) \\ &\quad \rightarrow CenarioCorrente(cen, t + 1)\end{aligned}$$

4.3.5 Movimentação e estado dos dragões

Um fator de complexidade no tratamento do movimento dos dragões é a direção do movimento, tendo em vista que os mesmos se movimentam em direção ao personagem. Dessa forma, definem-se as seguintes regras[3] para simplificar a obtenção da direção de movimento (ou seja, um vetor de coordenadas x e y) no plano cartesiano do objeto *o1* para o objeto *o2*:

[3] As três regras diferentes são necessárias pelo fato de que posições alinhadas no mesmo eixo causam divisão por zero.

- Quando os objetos estão alinhados em x:

$$\begin{aligned}\forall o1, o2, t\ (&\exists x1, y1\ Posicao(x1, y1, o1, t) \wedge \\ &\exists x2, y2\ Posicao(x2, y2, o2, t) \wedge \\ &x1 = x2\ \wedge\ \neg(y1 = y2)) \\ &\quad \rightarrow Direcao(o1, o2, 0, (y2 - y1)/Abs(y2 - y1), t)\end{aligned}$$

- Quando os objetos estão alinhados em y:

$$\begin{aligned}\forall o1, o2, t\ (&\exists x1, y1\ Posicao(x1, y1, o1, t) \wedge \\ &\exists x2, y2\ Posicao(x2, y2, o2, t) \wedge \\ &\neg(x1 = x2)\ \wedge\ y1 = y2) \\ &\quad \rightarrow Direcao(o1, o2, (x2 - x1)/Abs(x2 - x1), 0, t)\end{aligned}$$

- Quando os objetos não estão alinhados em nenhum eixo:

$$\begin{aligned}\forall o1, o2, t\ (&\exists x1, y1\ Posicao(x1, y1, o1, t) \wedge \\ &\exists x2, y2\ Posicao(x2, y2, o2, t) \wedge \\ &\neg(x1 = x2)\ \wedge\ \neg(y1 = y2)) \\ &\quad \rightarrow Direcao(o1, o2, (x2 - x1)/Abs(x2 - x1), (y2 - y1)/Abs(y2 - y1), t)\end{aligned}$$

Para que o dragão possa se movimentar, o mesmo não pode estar mordendo, e tanto ele quanto o personagem não podem estar mortos. Além disso, o dragão deve estar colidindo com o cenário corrente ou ser o dragão Yorgle (no qual tem acesso irrestrito a todos os cenários). Finalmente, pode-se definir a ação de *Movimentar* dos dragões pela sentença:

$$\begin{aligned}\forall d, t\ (&Dragao(d) \wedge \\ &\neg(\exists t1, t2\ Mordendo(d, t1, t2)\ \wedge\ t >= t1\ \wedge\ t <= t2) \wedge \\ &\neg Morto(d, t)\ \wedge\ \neg Morto(personagem, t) \wedge \\ &(\exists cen\ (CenarioCorrente(cen, t)\ \wedge\ Encosta(d, cen, t))\ \vee\ d = yorgle)) \wedge \\ &\exists dx, dy\ Direcao(d, personagem, dx, dy, t) \\ &\quad \rightarrow Movimentar(dx, dy, d, t)\end{aligned}$$

Caso exista, então, uma ação de *Movimentar*, a posição do dragão, bem como a posição da sua boca e barriga são alteradas através de:

$$\begin{aligned}\forall t\ (&\exists d\ Dragao(d)\ \wedge\ \exists drx, dry\ Posicao(drx, dry, d, t) \wedge \\ &\exists bc\ Boca(bc, d)\ \wedge\ \exists br\ Barriga(br, d) \wedge \\ &\exists dx, dy\ Movimentar(dx, dy, d, t) \wedge \\ &\quad \rightarrow Posicao(drx + dx, dry + dy, d, t + 1) \wedge \\ &\quad\ Posicao(drx + dx, dry + dy + 1, bc, t + 1) \wedge \\ &\quad\ Posicao(drx + dx + 1, dry + dy + 5, br, t + 1)\end{aligned}$$

Para os tempos em que não houve movimentação de algum dragão, a seguinte regra é necessária para definir que as posições não foram alteradas:

$$\begin{aligned}\forall t\ (&\exists d\ Dragao(d)\ \wedge\ \exists drx, dry\ Posicao(drx, dry, d, t)\ \wedge\\ &\exists bc\ Boca(bc, d)\ \wedge\ \exists br\ Barriga(br, d)\ \wedge\\ &\neg(\exists dx, dy\ Movimentar(dx, dy, d, t))\ \wedge\\ &\quad\rightarrow Posicao(drx, dry, d, t+1)\ \wedge\\ &\quad\ Posicao(drx, dry+1, bc, t+1)\ \wedge\\ &\quad\ Posicao(drx+1, dry+5, br, t+1)\end{aligned}$$

Para as posições da barriga e da boca do dragão foram usadas constantes pois elas estarão sempre a um deslocamento fixo do próprio dragão. Além disso, os dragões estarão sempre em algum dos estados: *normal*, *mordendo* ou *morto*. O estado *normal* fica ativo por padrão, e não necessita de uma regra de ativação (ela pode ser avaliada através da negação dos outros dois estados). O estado *mordendo* é ativado quando o dragão encosta no personagem, desde que já não esteja mordendo ou esteja morto. Esse estado é definido pela tupla *Mordendo*, a qual recebe como argumentos o dragão, o tempo que iniciou-se a mordida e o tempo em que ela terminará. Além disso, quando a ativação do estado de *mordendo* é realizada, o dragão se move instantaneamente para uma posição na qual o personagem ficará inserido em sua boca. A ocorrência do estado *mordendo* somente ocorrerá após a efetivação da ação *Morder*, que pode ser descrita pela sentença:

$$\begin{aligned}\forall t\ (&\exists d\ Dragao(d)\ \wedge\ \neg Morto(d, t)\ \wedge\ \neg Morto(personagem, t)\ \wedge\\ &\neg(\exists t1, t2\ Mordendo(d, t1, t2)\ \wedge\ t >= t1\ \wedge\ t <= t2)\ \wedge\\ &Encosta(d, personagem, t))\\ &\quad\rightarrow Morder(d, t)\end{aligned}$$

O dragão somente estará realmente mordendo pelo tempo determinado após a efetivação da ocorrência do estado *mordendo*, que é realizada através da tupla *Mordendo*:

$$\begin{aligned}\forall t\ (&\exists d\ Dragao(d)\ \wedge\ Morder(d, t)\ \wedge\\ &\exists drx, dry\ Posicao(drx, dry, personagem, t))\\ &\quad\rightarrow Mordendo(d, t+1, t+11)\ \wedge\ Posicao(drx, dry-1, d, t)\end{aligned}$$

O estado de *morto* de um dragão é ativado quando o mesmo não está morto e está encostando na espada. O estado de *morto* é permanente, ou seja, quando um dragão entra nesse estado, o mesmo permanecerá nele até o fim do jogo (essa regra serve igualmente para o personagem). As sentenças que descrevem esse procedimento são:

$$\begin{aligned}\forall t\ (&\exists d\ Dragao(d)\ \wedge\ \neg Morto(d, t)\ \wedge\\ &\neg(\exists t1, t2\ Mordendo(d, t1, t2)\ \wedge\ t >= t1\ \wedge\ t <= t2)\ \wedge\\ &\exists e\ Espada(e)\ \wedge\ Encosta(d, e, t))\\ &\quad\rightarrow Morto(d, t)\end{aligned}$$

e

$$\forall t\ \exists o\ Morto(o, t) \rightarrow Morto(o, t+1)$$

4.3.6 Morte do personagem

O personagem será engolido e, consequentemente, morto, quando estiver encostando em um dragão, e esse terminar sua mordida. Isso gera a ação de *Engolir*:

$$\begin{aligned}\forall t\ (\exists d\ Dragao(d)\ \wedge\ \exists t1, t2\ Mordendo(d, t1, t2)\ \wedge\ \neg Morto(d, t)\ \wedge \\ \exists bc\ Boca(bc, d)\ \wedge\ Encosta(bc, personagem, t)\ \wedge\ t = t2 \\ \rightarrow Morto(personagem, t)\ \wedge \\ Engolir(d, personagem, t)\end{aligned}$$

A ação de engolir, quando efetivada, posicionará o personagem dentro da barriga do dragão. A regra correspondente é definida como:

$$\begin{aligned}\forall t\ (\exists d\ Dragao(d)\ \wedge\ Engolir(d, personagem, t)\ \wedge \\ \exists br\ Barriga(br, d)\ \wedge\ \exists brx, bry\ Posicao(brx, bry, br, t)) \\ \rightarrow Posicao(brx, bry, personagem, t+1)\end{aligned}$$

4.3.7 Abertura dos portões

Por padrão, todos os portões estarão fechados. Quando uma chave encostar no portão de um castelo de cor corresponde, este se abrirá. Isso se dará pela tupla *Aberto* através da regra:

$$\begin{aligned}\forall t\ (\exists ch, cor\ Item(ch)\ \wedge\ Chave(ch, cor)\ \wedge \\ \exists po\ Portao(po, cor)\ \wedge\ Encosta(ch, po, t)\ \wedge \\ \neg Aberto(po, t) \\ \rightarrow Aberto(po, t+1)\end{aligned}$$

O estado de *Aberto* para o portão deverá persistir nos tempos posteriores, e é definida pela regra:

$$\forall t\ (\exists po, cor\ Portao(po, cor)\ \wedge\ Aberto(por, t))\ \rightarrow\ Aberto(por, t+1)$$

4.3.8 Fim de jogo

O fim de jogo se dá quando o cálice se encontra no cenário correspondente ao interior do castelo amarelo. Para tanto, será utilizada a tupla *FimJogo* para definir que no tempo t o jogo terminou:

$$\begin{aligned}\exists t\ (\exists ca\ Item(ca)\ \wedge\ Calice(ca)\ \wedge \\ \exists cen\ CenarioCorrente(cen, t)\ \wedge\ cen = casteloAmarelo\ \wedge \\ \neg FimJogo(t)\ \wedge\ ContemObjeto(cen, ca, t) \\ \rightarrow FimJogo(t+1)\end{aligned}$$

O fim de jogo é permanente, ou seja, uma vez que acontece, estará sempre ativo nos tempos seguintes:

$$\exists t\ FimJogo(t)\ \rightarrow\ FimJogo(t+1)$$

5 CONSTRUÇÃO DA APLICAÇÃO

O protótipo do jogo Adventure foi desenvolvido utilizando uma interface que foi desenvolvida na linguagem C# (GREENE; STELLMAN, 2013), que possui o paradigma orientado a objetos, integrada com uma base de conhecimento em Prolog. Essa aplicação tem como responsabilidade realizar os tratamentos de entrada e saída de dados através do *framework* XNA, ferramenta para auxílio de desenvolvimento de jogos da Microsoft (MICROSOFT, 2014). A aplicação em C# também é responsável por efetuar a comunicação com a base de conhecimento em Prolog. Essa comunicação é realizada através da interface SwiPlCs (LESTA, 2014), que tem como funcionalidade carregar uma base de conhecimento desenvolvida em Prolog, e mantê-la aberta para consultas.

5.1 Base de Conhecimento

A base de conhecimento foi construída em Prolog, que é uma linguagem de programação declarativa utilizada para resolver problemas que envolvem objetos e seus relacionamentos (CLOCKSIN; MELLISH, 1994). Um arquivo em Prolog possui a extensão *.pl*, e consiste em uma lista de fatos e regras. A utilização da linguagem consiste em três passos:

1. Declaração de fatos: declaram-se na base de conhecimento os relacionamentos que são fatos. Por exemplo, podemos dizer que a posição no tempo *0* do *personagem* é de [10,10] através do fato: `posicao(10,10,personagem,0).`;

2. Declaração de regras: declaram-se as regras que atuam como condições para criação de relacionamentos. As variáveis da regra são representadas por palavras com a letra inicial em maiúsculo. Por exemplo, a regra para carregar um item é definida como:

   ```
   carregar(I):- item(I),encosta(personagem,I).
   ```

   Nessa regra, a tupla **carregar** que antecede o lexema `:-` é o resultado da avaliação das premissas **item** e **encosta** (a vírgula é o operador de conjunção em Prolog);

3. Consultas: executam-se consultas à base de conhecimento sobre informações de interesse. As consultas são realizadas através de tuplas utilizando variáveis cujo valor é desconhecido. Os valores são retornados pela base nas variáveis recebidas como argumento. Por exemplo, podemos querer tomar conhecimento

da posição do personagem $[X, Y]$ no tempo 5. Para isso, a seguinte consulta pode ser executada na base de conhecimento: **posicao**(X,Y,personagem,5)., sendo que as coordenadas do personagem serão retornadas nas variáveis X e Y.

Nas Seções 5.1.1 e 5.1.2 é apresentada a forma como foram declarados os fatos para o protótipo do jogo Adventure. Já na Seção 5.1.3 é explicado de que maneira as regras da ontologia em lógica de primeira ordem foram transcritas para a linguagem Prolog.

5.1.1 Declaração de Fatos correspondentes aos Cenários

A declaração dos fatos correspondentes aos cenários foi dividida de acordo com o cenário, ou seja, cada um deles possui um arquivo, o qual descreve as posições e tamanhos de cada parede, portão e transição contidos neste cenário. Os cenários foram nomeados pela palavra *Cenario* seguida de um sufixo inteiro de dois dígitos, sem ordem específica[1]. Já as paredes foram nomeadas com o prefixo *par*, os portões *por* e as transições *tr*, todos com um sufixo enumerador de três, dois e dois dígitos, respectivamente. O mapeamento dos cenários e seus objetos foi feito conforme demonstra a Figura 5.1.

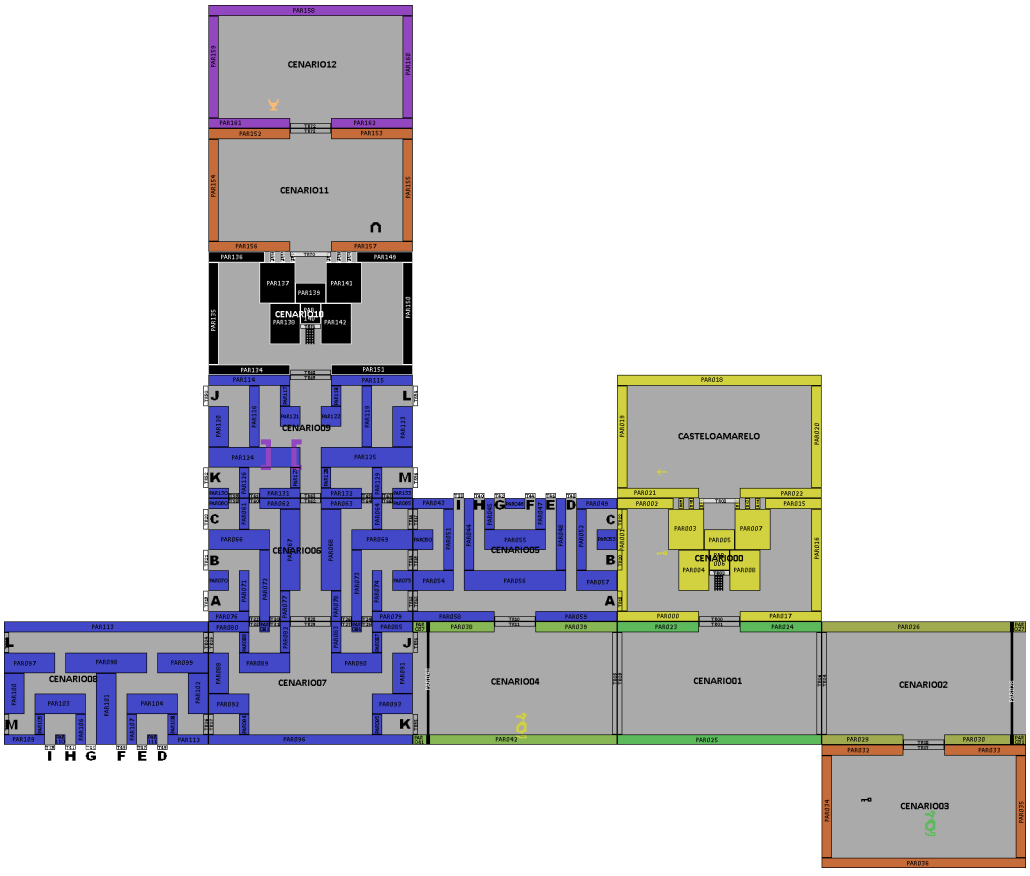


Figura 5.1: Mapeamento dos cenários, paredes, portões e transições do jogo.

[1]Os arquivos de nome Cenario00 até Cenario12 compõem treze cenários, de um total de quatorze. A exceção é o castelo amarelo, no qual foi nomeado *CasteloAmarelo* devido a sua importância na regra de fim de jogo (vide Seção 4.3.8).

Por exemplo, a parede *par000* do cenário *cenario00* foi inserida na base de conhecimento como:

```
parede(par000, cenario00).
retangulo(par000, 128, 16).
posicao(960, 949, par000, 0).
```

5.1.2 Declaração de Fatos correspondentes ao Personagem, Itens e Dragões

Além das posições e tamanhos dos objetos dos cenários, foi necessário definir a posição inicial do personagem, dos itens e dos dragões, bem como seus tamanhos e estados. Essas inicializações foram feitas no arquivo principal *Adventure.pl*. A inicialização do personagem, por exemplo, foi realizada através dos fatos:

```
retangulo(personagem, 8, 8).
posicao(1116, 918, personagem, 0).
morto(personagem, 0, 0).
```

5.1.3 Definição das Regras do Jogo

A passagem das regras de lógica em primeira ordem para Prolog é simples. As tuplas e variáveis da lógica se tornam as tuplas e variáveis do Prolog. Por exemplo, a regra a seguir define que toda chave, espada ou cálice é um item

$$\forall i \ (\exists cor \ Chave(i, cor)) \ \vee \ Espada(i) \ \vee \ Calice(i) \ \rightarrow Item(i)$$

Essa regra pode ser facilmente transcrita para Prolog transformando, por exemplo, $Chave(i, cor)$ em `chave(I,_)`. O caractere de traço inferior tem função de representar uma variável anônima, utilizada para quando não se faz necessário conhecer seu valor. Como no caso da regra acima, a regra não difere de acordo com a cor da chave, pois qualquer chave é um item, independente de sua cor. A regra completa, portanto, foi transcrita como:

```
item(I) :- chave(I, _); espada(I); calice(I).
```

A transcrição do quantificador existencial na transcrição acontece de forma direta. Quando informamos uma consulta a base de conhecimento em Prolog, como por exemplo `chave(I,COR)`, ele procurará pela **existência** de uma ocorrência da tupla. Isso também acontece quando a consulta está contida em uma regra (como na anterior). Portanto, o quantificador existencial está compreendido implicitamente na linguagem Prolog.

Quanto ao quantificador universal, existem duas situações. Quando ele abrange uma regra inteira, como no caso da regra que define um item, o quantificador $\forall$item como objetivo abranger todos objetos do universo em questão. Esse comportamento não faz sentido em Prolog, uma vez que toda regra declarada na base de conhecimento pode ser usada para qualquer consulta necessária. Portanto, nesses casos, o quantificador universal foi ignorado. A segunda situação ocorre quando o quantificador universal é inserido em uma posição interna da regra, e possui como objetivo encontrar todas as instâncias que atendam uma certa condição. A linguagem Prolog já disponibiliza a tupla **forall**, que possui um comportamento equivalente para essas situações. Seu propósito é retornar se todas as ocorrências possíveis de uma subconsulta (consulta que faz parte de outra consulta) são verdadeiras. A regra que

gera a tupla *PodeMovimentar* faz uso dessa funcionalidade, pois ela tem como responsabilidade verificar, para todas paredes contidas no cenário corrente, se o jogador não as encosta.

Observou-se perda de desempenho em algumas regras. Por exemplo, ao realizar uma consulta da tupla *FimJogo* no tempo *10*, o motor do Prolog teria que, primeiramente, verificar se houve uma ocorrência desta no tempo *10*. Em caso negativo, o motor verificaria a existência de *FimJogo* nos tempos anteriores. Ou seja, verificaria a existência da tupla no tempo *9*, e, em caso negativo, nos tempos *8*, *7*, *6* e assim por diante, até o tempo *0*, se necessário. Portanto, uma consulta dessas tem complexidade O(n), isto é, uma consulta no tempo *100* geraria outras *100* consultas, e uma consulta no tempo *1000* geraria outras *1000* consultas.

Portanto, as tuplas que tratam de estados que persistem em várias instâncias de tempo sofreram modificações para melhorar o desempenho. Então, para o exemplo anterior por exemplo, em toda instância de tempo anterior à atual deve existir uma ocorrência da tupla *FimJogo*, independente de seu valor. Consequentemente, a tupla necessita de um novo parâmetro, que tem como objetivo informar qual é o estado de *FimJogo*: falso (número 0), ou verdadeiro (número 1). Por exemplo, uma regra que utiliza a tupla de fim de jogo foi definida como:

```
fimJogo(TF, S) :- T is TF - 1, once(fimJogo(T, S)).
```

Além disso, alterações foram feitas também para certas regras, como o exemplo da regra que gera a tupla *Encosta* definida por

$$\begin{aligned}\forall o1, x1, y1, lar1, alt1,\ (Retangulo(o1, lar1, alt1)\ \wedge\ Retangulo(o2, lar2, alt2)\ \wedge \\ o2, x2, y2, lar2, alt2, t\ Posicao(x1, y1, o1, t)\ \wedge\ Posicao(x2, y2, o2, t)\ \wedge \\ Colide(x1, y1, lar1, alt1, x2, y2, lar2, alt2)) \\ \rightarrow Encosta(o1, o2, t)\end{aligned}$$

que foi transcrita em Prolog como:

```
encosta(O1, O2, T) :-
     retangulo(O1, Lar1, Alt1),
     retangulo(O2, Lar2, Alt2),
     posicao(X1, Y1, O1, T),
     posicao(X2, Y2, O2, T),
     !,
     colide(X1, Y1, Lar1, Alt1, X2, Y2, Lar2, Alt2).
```

O que ocorre nessa regra é que o motor do Prolog, ao não encontrar resultado para a tupla **colide**, efetuaria um *backtracking*, ou seja, procuraria outro resultado para as variáveis `X2` e `Y2` na entrada `posicao(X2, Y2, O2, T)`. Entretanto, o motor não encontrará essa outra ocorrência, pois sabe-se que um objeto na base de conhecimento possuirá somente uma única posição em determinado tempo. Por isso, utilizou-se o comando *cut*, que é identificado pelo caractere de ponto de exclamação, para dizer que a partir deste ponto o *backtracking* não é mais necessário (CLOCKSIN; MELLISH, 1994). Portanto, algumas das regras, quando convertidas para Prolog, sofreram modificações baseadas na premissa de que somente uma instância da tupla existirá ao mesmo tempo na base de conhecimento.

As tuplas de estados também seguem a premissa de unicidade. Por exemplo, a tupla *FimJogo* deverá existir uma única vez para cada tempo. Dessa forma, utiliza-

se a tupla **once** da linguagem Prolog. Essa permite que o motor não procure por mais resultados, e considere somente o primeiro encontrado.

Outras alterações nas transcrições das regras ocorreram naquelas que definem informações para o tempo seguinte. Uma delas é a de posição futura do personagem enquanto o mesmo está parado, onde diz-se que a posição do personagem que não se movimentou, nem encostou em transições e não foi engolido estará no tempo *t+1* na mesma posição que estava em *t*. Essa regra foi descrita por

$$\begin{aligned}
&\forall t\ (\exists x, y\ Posicao(x, y, personagem, t) \wedge \\
&\quad (FimJogo(t) \vee \\
&\quad \neg(\exists dx, dy\ Movimentar(dx, dy, personagem, t)) \vee \\
&\quad \neg PodeMovimentar(personagem, t)) \wedge \\
&\quad \neg(\exists tx, ty\ Transicionar(tx, ty, personagem, t)) \wedge \\
&\quad \neg(\exists d\ Dragao(d)\ \wedge\ Engolir(d, personagem, t))) \\
&\qquad\qquad \rightarrow Posicao(x, y, personagem, t + 1)
\end{aligned}$$

e uma possível conversão dela para Prolog pode ser definida como:

```
posicao(X, Y, personagem, TF) :-
     (once(fimJogo(T, S)),
          S == 1;
          not(movimentar(_, _, personagem, T));
          not(podeMovimentar(personagem, T))),
     not(transicionar(_, _, personagem, _, T)),
     not((engolir(D, personagem, T),
     dragao(D))),
     posicao(X, Y, personagem, T),
     TF is T + 1.
```

Um problema com essa regra é de que motor do Prolog, ao tentar resolver uma consulta de posição no tempo *10*, ou seja, `posicao(X, Y, personagem, 10)`, associaria a variável `TF` (que representa o tempo fim) ao valor *10*. Como o motor do Prolog desconhece o valor da variável `T`, e as subconsultas que compõem a regra fazem uso dessa variável, a condição `TF is T + 1` será, na maioria das vezes, considerada falsa, uma vez que, como no exemplo, o único valor válido para ela é *9*. Portanto, o valor de T pode ser definido no início da regra, removendo `TF is T + 1` do fim da regra e adicionando ao início a condição `T is TF - 1`.

5.2 Desenvolvimento da Aplicação

A aplicação do jogo, responsável pela comunicação com a base de conhecimento em Prolog e realizar o tratamento de entrada e saída foi desenvolvido na linguagem C# utilizando XNA, que é um *framework* da Microsoft que disponibiliza funcionalidades específicas para o desenvolvimento de jogos. Esse *framework* tem como finalidade desenhar o conteúdo gráfico dos objetos do jogo (cenários, itens, dragões, etc) na tela. Além disso, é responsável por fazer o reconhecimento do pressionamento das setas direcionais (para controlar a movimentação) e da barra de espaço (para largar itens), além das teclas F1 para reiniciar o jogo e F2 para salvar em um arquivo (de nome *database.pl*) o estado atual da base de conhecimento.

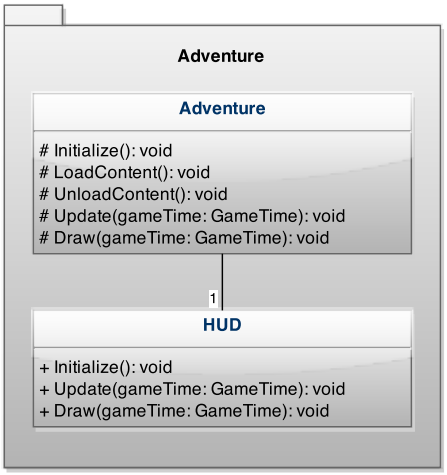


Figura 5.2: Diagrama de classes do pacote Adventure da aplicação.

O diagrama de classes correspondente à aplicação pode ser visualizado na Figura 5.2, que contém as classes básicas do jogo. Outras classes podem ser visualizadas na Figura **??**, localizada no Apêndice B. A classe *Adventure*, que herda da classe *Game* do XNA, é responsável pelo tratamento das funcionalidades básicas do jogo, como o carregamento e descarregamento de recursos (imagens, fontes, etc) e inicialização de classes. Um dos métodos dessa classe é o *Update*, o qual é executado sessenta vezes por segundo pelo *framework*, e é o ponto onde a lógica do jogo deve ser desenvolvida. Portanto, este é o local onde as consultas na base de conhecimento em Prolog foram efetuadas e posteriormente armazenadas em atributos em suas respectivas classes. Com os valores armazenados, o método *Draw*, que é sincronizado com o método *Update*, é executado para desenhar os objetos em suas posições e estados específicos. A aplicação em C# também foi responsável pelo gerenciamento da variável que controla o tempo, e que tem seu valor incrementado a cada fim do método *Update*.

A comunicação com a base de conhecimento em Prolog foi feita através da interface SwiPlCs. Por intermédio dela, é possível carregar um arquivo Prolog e fazer consultas sobre o mesmo. A base de conhecimento do Adventure foi inicializada através da linha de código existente no quadro:

```
1 PlEngine.Initialize(new[] { "-q", @"Prolog\Adventure.pl" });
```

Já no seguinte quadro têm-se as linhas de código que representam o procedimento referente a atualização da posição do personagem:

```
1 using (var positionsQuery = new
     PlQuery(string.Format("posicao(X, Y, personagem, {0})",
     time)))
2 {
3         var position = positionsQuery.SolutionVariables.First();
4         var x = Convert.ToInt32(position["X"].ToString()) -
             (int)currentScenario.Offset.X;
5         var y = Convert.ToInt32(position["Y"].ToString()) -
             (int)currentScenario.Offset.Y;
```

```
6           _bounds = new Rectangle(x, y, _bounds.Width,
                _bounds.Height);
7   }
```

Na linha 1, inicia-se um objeto para a consulta **posicao**(X,Y,personagem,{0}), sendo {0} o tempo atual. Na linha 3, é retornado o primeiro resultado da consulta. Já nas linhas 4 e 5, os valores são convertidos para inteiro. Finalmente, na linha 6, os valores inteiros são armazenados em um atributo da classe.

Como visto anteriormente (na Seção 5.1.3), algumas regras apresentaram problemas de desempenho, tal como a consulta da tupla *FimJogo*. No caso destas regras, após consultar as informações dos objetos em determinado tempo *t*, a aplicação insere-as na base de conhecimento em forma de fatos. Dessa forma, nas consultas para *t+1*, o motor somente precisaria regressar até o tempo *t*, e, a partir desse ponto, calcular a informação para *t+1*. Essa inserção na base de conhecimento foi feita através das tuplas **asserta** e **retract**. A primeira é responsável por adicionar um fato à base de conhecimento, inserindo-o no início do arquivo. Já a segunda foi utilizada para remover fatos antigos e assim evitar o crescimento da base de conhecimento. Por exemplo, quando as informações do tempo *t+1* são inseridas na base de conhecimento, as informações do tempo *t* são removidas.

Na Figura 5.3, tem-se uma imagem animada do jogo (visualizável através do arquivo em formato *pdf* - *portable document format* - disponível no CD em anexo). Essa imagem demonstra uma partida completa do jogo Adventure gerada a partir do protótipo construído. Nela, o personagem obtém a espada e mata ambos os dragões. Depois, atravessa o labirinto e apanha o cálice utilizando a chave preta para abrir o castelo preto. Por fim, o jogador retorna ao castelo amarelo detendo o cálice e, assim, a partida termina.

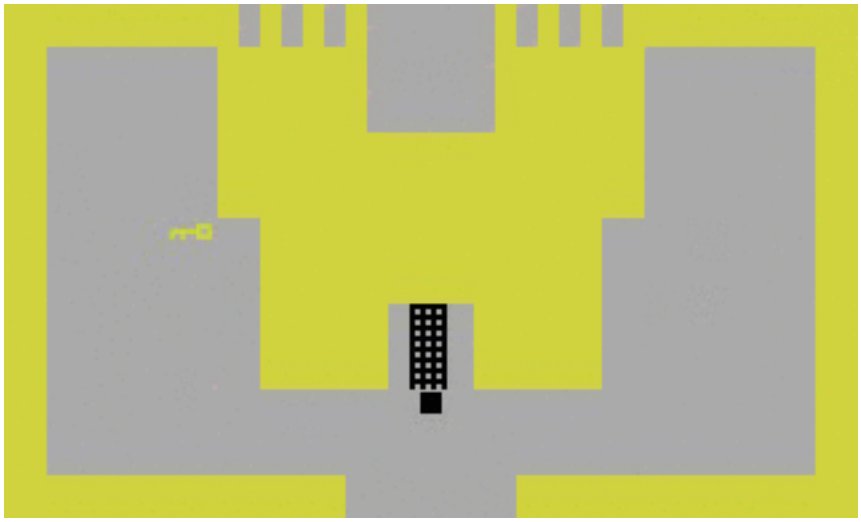


Figura 5.3: Imagem animada de uma partida completa do jogo Adventure.

6 CONSIDERAÇÕES FINAIS

Neste trabalho foi desenvolvido um protótipo do jogo Adventure do console Atari 2600 separando, de forma explícita, as regras do jogo de outros recursos computacionais necessários para uma aplicação. Para tanto, foi utilizada uma ontologia para a formalização das regras do jogo. Posteriormente, essas regras foram transcritas na linguagem Prolog e, por fim, uma aplicação em C# foi desenvolvida para realizar a comunicação com essa base de conhecimento e desempenhar os tratamentos de entrada e saída.

A escolha da lógica de primeira ordem como forma de formalização da ontologia mostrou-se adequada para a representação das regras do jogo Adventure. Isso se deve, principalmente, pela sua flexibilidade e simplicidade, elementos que também auxiliam na leitura e compreensão futuras das sentenças construídas.

Apesar de algumas sentenças terem se tornado extensas, essas poderiam facilmente serem divididas em sentenças menores, de leitura mais fácil. Mesmo que algumas regras do jogo original tenham sido desconsideradas para a construção do protótipo, a ontologia em lógica de primeira ordem desenvolvida abrange um conjunto suficiente de regras e, inclusive, pode servir como uma base consistente para possíveis desenvolvimentos futuros de outros modos do jogo Adventure.

Já na transcrição das regras para Prolog, algumas alterações tiveram de ser efetuadas para tratar de certas características da linguagem. Um dos exemplos é a remoção do quantificador universal em determinadas regras. Além disso, foi necessário adicionar tratamentos para casos em que o desempenho do jogo diminui. Nestes casos, os fatos foram introduzidos na base de conhecimento de forma a contornar essa limitação computacional.

De modo geral, o resultado final do trabalho se mostrou interessante. Verificou-se que sim, é possível planejar o desenvolvimento de um jogo utilizando uma ontologia, de forma a separar as regras do jogo da interface. A ontologia formalizada do jogo Adventure ficou de fácil leitura e alteração. Isso ocorreu devido a clareza das tuplas, no qual a regra para algum comportamento específico pode ser encontrada rapidamente na ontologia. Além disso, a transcrição das regras para Prolog mostrou-se pouco complexa, apesar de ter necessitado de tratamento para algumas exceções. Por fim, a interface para integração com a base de conhecimento mostrou-se de fácil utilização e simples na execução de consultas.

6.1 Trabalhos Futuros

Adventure é um jogo de regras simples, e já é superado por muitas gerações de jogos, que são maiores e mais complexos em comparação com os da época do

Atari. Por isso, um futuro trabalho poderia incluir um estudo sobre a utilização de ontologias em jogos de grande porte.

Além disso, é possível continuar a implementação do jogo Adventure para os outros modos de jogo para verificar se emergem problemas na transcrição para Prolog. Pode-se verificar, também, a utilização de outras formas de formalização da ontologia, de modo a facilitar essa transcrição entre lógica e linguagem.

REFERÊNCIAS

APPERLEY, T. H. Genre and game studies: toward a critical approach to video game genres. **Simulation & Gaming**, [S.l.], v.37, n.1, p.6–23, 2006.

ARRUDA, E. P. **Fundamentos para o Desenvolvimento de Jogos Digitais**. [S.l.]: Bookman, 2014.

ATARI. **Adventure**: game program instructions. [S.l.: s.n.], 1980.

CLOCKSIN, W.; MELLISH, C. **Programming in Prolog**. [S.l.]: Springer Berlin Heidelberg, 1994.

CORCHO, O.; FERNÁNDEZ-LÓPEZ, M.; GÓMEZ-PÉREZ, A. Methodologies, tools and languages for building ontologies. Where is their meeting point? **Data & knowledge engineering**, [S.l.], v.46, n.1, p.41–64, 2003.

EDERY, D.; MOLLICK, E. **Changing the game**: how video games are transforming the future of business. [S.l.]: FT Press, 2008.

FILHO, P.; HETEM, A. **Lógica para Computação**. [S.l.]: LTC, 2012.

GIARETTA, P.; GUARINO, N. Ontologies and knowledge bases towards a terminological clarification. **Towards very large knowledge bases: knowledge building & knowledge sharing**, [S.l.], v.25, 1995.

GREENE, J.; STELLMAN, A. **Head First C#**. [S.l.]: O'Reilly Media, 2013. (Head first series).

HECK, R.; MAY, R. Truth in Frege. **Oxford Handbook of Truth**, [S.l.], 2013.

HOCHHALTER, B. et al. Towards an Ontological Language for Game Analysis. **DiGRA**, [S.l.], 2005.

LESTA, U. **SwiPlCs**. [S.l.: s.n.], 2014. <Disponível em: `http://www.swi-prolog.org/contrib/CSharp.html`>. Acesso em: 15 de abr. de 2014.

MACHADO, A. F. d. V. **Uma engine em XNA e PROLOG para apoio ao ensino de programação declarativa**. 2009. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal Fluminense.

MADEIRA, C. A. G. **FORGE V8**: um framework para o desenvolvimento de jogos de computador e aplicações multimídia. 2001. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal de Pernambuco.

MICROSOFT. **XNA Game Studio**. [S.l.: s.n.], 2014. <Disponível em: `https://msxna.codeplex.com/`>. Acesso em: 20 de nov. de 2014.

MORA, J.; TERRICABRAS, J. **Diccionario de filosofía**. [S.l.]: Editorial Ariel, S.A., 1994. n.v. 3. (Ariel referencia).

MUSEN, M. A. Ontology-oriented design and programming. **Knowledge Engineering and Agent Technology**, [S.l.], v.52, p.3–16, 2000.

NECHES, R. et al. Enabling technology for knowledge sharing. **AI magazine**, [S.l.], v.12, n.3, p.36, 1991.

RUSSELL, S. J. et al. **Artificial intelligence**: a modern approach. [S.l.]: Prentice hall Englewood Cliffs, 2010.

SILVA, F. C. L. da. **Uma ferramenta para o ensino de inteligência artificial usando jogos de computador**. 2007. Dissertação (Mestrado em Ciência da Computação) — Universidade de São Paulo.

STUDER, R.; BENJAMINS, V. R.; FENSEL, D. Knowledge engineering: principles and methods. **Data & knowledge engineering**, [S.l.], v.25, n.1, p.161–197, 1998.

VAREJÃO, F. **Linguagens de programação**: JAVA, C e C++ e outras. [S.l.]: Editora Campus, 2004.

ZAFEIROPOULOS, V. **Adventure games implementation under the Prolog programming language.** 2008. Dissertação (Mestrado em Ciência da Computação) — Mälardalen University.

APÊNDICES A

Aqui foram listadas, de forma ininterrupta, as regras em lógica de primeira ordem que foram definidas no Capítulo 4 e que compõem a ontologia criada neste trabalho.

- Definição dos itens:

$$\forall i\ (\exists cor\ Chave(i, cor))\ \vee\ Espada(i)\ \vee\ Calice(i)\ \rightarrow Item(i)$$

- Colisão de retângulos:

$$\forall x1, y1, lar1, alt1, x2, y2, lar2, alt2\ (\neg(x1 > x2 + lar2 - 1\ \vee\ x1 + lar1 - 1 < x2\ \vee \\ y1 > y2 + alt2 - 1\ \vee\ y1 + alt1 - 1 < y2)) \\ \rightarrow Colide(x1, y1, lar1, alt1, x2, y2, lar2, alt2)$$

- Colisão de objetos:

$$\forall o1, x1, y1, lar1, alt1,\ (Retangulo(o1, lar1, alt1)\ \wedge\ Retangulo(o2, lar2, alt2)\ \wedge \\ o2, x2, y2, lar2, alt2, t\ Posicao(x1, y1, o1, t)\ \wedge\ Posicao(x2, y2, o2, t)\ \wedge \\ Colide(x1, y1, lar1, alt1, x2, y2, lar2, alt2)) \\ \rightarrow Encosta(o1, o2, t)$$

- Colisão de objetos com posições de teste:

$$\forall o1, x1, y1, lar1, alt1,\ (Retangulo(o1, lar1, alt1)\ \wedge\ Retangulo(o2, lar2, alt2)\ \wedge \\ o2, x2, y2, lar2, alt2, t\ PosicaoTeste(x1, y1, o1, t)\ \wedge\ PosicaoTeste(x2, y2, o2, t)\ \wedge \\ Colide(x1, y1, lar1, alt1, x2, y2, lar2, alt2)) \\ \rightarrow EncostaTeste(o1, o2, t)$$

- Geração da posição de teste:

$$\forall o, t\ (\exists dx, dy\ Movimentar(dx, dy, o, t)\ \wedge \\ \exists x, y\ Posicao(x, y, o, t)) \\ \rightarrow PosicaoTeste(x + dx, y + dy, o, t)$$

e

$$\forall o, t\ (\neg\exists dx, dy\ Movimentar(dx, dy, o, t)\ \wedge \\ \exists x, y\ Posicao(x, y, o, t)) \\ \rightarrow PosicaoTeste(x, y, o, t)$$

QUERO QUE GERE ERRO AQUI FALAR DA SEÇÃO NA PÁGINA 39 NO E-MAIL PARA O PROFESSOR!

- Verificação de um retângulo que contém outro:

$$\forall x1, y1, lar1, alt1, x2, y2, lar2, alt2\ (x1 <= x2\ \wedge\ x1 + lar1 - 1 >= x2 + Lar2 - 1 \wedge y1 <= y2\ \wedge\ y1 + alt1 - 1 >= y2 + Alt2 - 1) \rightarrow Contem(x1, y1, lar1, alt1, x2, y2, lar2, alt2)$$

- Verificação de um objeto que contém outro:

$$\forall o1, x1, y1, lar1, alt1,\ (Retangulo(o1, lar1, alt1)\ \wedge\ Retangulo(o2, lar2, alt2) \wedge o2, x2, y2, lar2, alt2, t\ Posicao(x1, y1, o1, t)\ \wedge\ Posicao(x2, y2, o2, t) \wedge Contem(x1, y1, lar1, alt1, x2, y2, lar2, alt2)) \rightarrow ContemObjeto(o1, o2, t)$$

- Verificação de um objeto que contém outro com posições de teste:

$$\forall o1, x1, y1, lar1, alt1,\ (Retangulo(o1, lar1, alt1)\ \wedge\ Retangulo(o2, lar2, alt2) \wedge o2, x2, y2, lar2, alt2, t\ PosicaoTeste(x1, y1, o1, t)\ \wedge\ PosicaoTeste(x2, y2, o2, t) \wedge Contem(x1, y1, lar1, alt1, x2, y2, lar2, alt2)) \rightarrow ContemObjetoTeste(o1, o2, t)$$

- Efetivação da movimentação do personagem:

$$\forall t\ (\exists x, y\ Posicao(x, y, personagem, t) \wedge \exists dx, dy\ Movimentar(dx, dy, personagem, t) \wedge PodeMovimentar(personagem, t) \wedge \neg FimJogo(t) \wedge \neg(\exists tx, ty\ Transicionar(tx, ty, personagem, t)) \wedge \neg(\exists d\ Dragao(d)\ \wedge\ Engolir(d, personagem, t))) \rightarrow Posicao(x + dx, y + dy, personagem, t + 1)$$

- Validação de movimentação do personagem:

$$\forall o, t\ \neg(\exists c, par\ CenarioCorrente(c, t)\ \wedge\ Parede(par, c)\ \wedge\ EncostaTeste(par, o, t)) \wedge (\forall d\ Dragao(d)\ \wedge\ \neg EncostaTeste(d, o, t) \vee (\neg Morto(d, t)\ \wedge\ Mordendo(d, t1, t2)\ \wedge\ \exists bc\ Boca(bc, d) \wedge (ContemObjetoTeste(bc, o, t) \vee (EncostaTeste(bc, o, t)\ \wedge\ \neg ContemObjetoTeste(d, o, t))))) \neg(\exists po, cor\ Portao(po, cor)\ \wedge\ \neg Aberto(po, t)\ \wedge\ EncostaTeste(po, o, t)) \rightarrow PodeMovimentar(o, t)$$

- Definição da posição do personagem imóvel:

$$\begin{aligned}
&\forall t\ (\exists x, y\ Posicao(x, y, personagem, t) \wedge \\
&\quad (FimJogo(t) \vee \\
&\quad \neg(\exists dx, dy\ Movimentar(dx, dy, personagem, t)) \vee \\
&\quad \neg PodeMovimentar(personagem, t)) \wedge \\
&\quad \neg(\exists tx, ty\ Transicionar(tx, ty, personagem, t)) \wedge \\
&\quad \neg(\exists d\ Dragao(d)\ \wedge\ Engolir(d, personagem, t))) \\
&\qquad\qquad \rightarrow Posicao(x, y, personagem, t + 1)
\end{aligned}$$

- Geração da ação de carregamento de itens:

$$\forall t\ \exists i\ Item(i)\ \wedge\ Encosta(personagem, i, t) \rightarrow Carregar(i, t)$$

- Efetivação da ação de carregamento de item:

$$\forall t\ \exists i\ Item(i)\ \wedge\ Carregar(i, t) \rightarrow Carregando(i, t + 1)$$

- Permanência do carregamento de itens:

$$\begin{aligned}
&\forall t\ (\exists i\ Item(i)\ \wedge\ Carregando(i, t)\ \wedge\ \neg Largar(t) \wedge \\
&\quad \neg(\exists i2\ Item(i2)\ \wedge\ Carregar(i2, t)\ \wedge\ \neg(i = i2))) \\
&\qquad\qquad \rightarrow Carregando(i, t + 1)
\end{aligned}$$

- Definição da posição dos itens:

$$\begin{aligned}
&\forall t\ (\exists i\ Item(i)\ \wedge\ Carregando(i, t) \wedge \\
&\quad \exists x, y, lar, alt\ Posicao(x, y, personagem, t)\ \wedge\ Retangulo(i, lar, alt)) \\
&\qquad \rightarrow Posicao(x, y - alt - 12, i, t)
\end{aligned}$$

- Definição da posição dos itens que estão imóveis

$$\begin{aligned}
&\forall t\ \exists i\ Item(i)\ \wedge\ \neg Carregando(i, t)\ \wedge\ Posicao(x, y, i, t) \\
&\qquad \rightarrow Posicao(x, y, i, t)
\end{aligned}$$

- Ativação de transição horizontal:

$$\begin{aligned}
&\forall t\ (\exists tr, tpx, tpy, cen\ TransicaoHorizontal(tr, tpx, tpy, cen) \wedge \\
&\quad \neg CenarioCorrente(cen, t) \wedge \\
&\quad Encosta(tr, personagem, t)) \wedge \\
&\quad \exists tx, ty\ Posicao(tx, ty, tr, t)\ \wedge\ \exists px, py\ Posicao(px, py, personagem, t) \\
&\qquad\qquad \rightarrow Transicionar(tpx, tpy + py - ty, personagem, t) \wedge \\
&\qquad\qquad TrocarCenarioCorrente(cen, t)
\end{aligned}$$

- Ativação de transição vertical:

$$\begin{aligned}\forall t\ (&\exists tr, tpx, tpy, cen\ TransicaoVertical(tr, tpx, tpy, cen) \wedge \\ &\neg CenarioCorrente(cen, t) \wedge \\ &Encosta(tr, personagem, t)) \wedge \\ &\exists tx, ty\ Posicao(tx, ty, tr, t)\ \wedge\ \exists px, py\ Posicao(px, py, personagem, t) \\ &\qquad \rightarrow Transicionar(tpx + px - tx, tpy, personagem, t) \wedge \\ &\qquad\ TrocarCenarioCorrente(cen, t)\end{aligned}$$

- Ativação de transição de posição:

$$\begin{aligned}\forall t\ (&\exists tr, tpx, tpy, cen\ TransicaoPosicao(tr, tpx, tpy, cen) \wedge \\ &\neg CenarioCorrente(cen, t)) \wedge \\ &Encosta(tr, personagem, t) \\ &\qquad \rightarrow Transicionar(tpx, tpy, personagem, t) \wedge \\ &\qquad\ TrocarCenarioCorrente(cen, t)\end{aligned}$$

- Efetivação de transição:

$$\begin{aligned}\forall t\ (&\exists tpx, tpy\ Transicionar(tpx, tpy, personagem, t) \wedge \\ &\neg(\exists d\ Dragao(d)\ \wedge\ Engolir(d, personagem, t))) \\ &\qquad \rightarrow Posicao(tpx, tpy, personagem, t + 1)\end{aligned}$$

- Definição do cenário corrente:

$$\forall t\ \exists cen\ TrocarCenarioCorrente(cen, t) \rightarrow CenarioCorrente(cen, t + 1)$$

e

$$\begin{aligned}\forall t\ (&\exists cen\ CenarioCorrente(cen, t) \wedge \\ &\neg(\exists cen2\ TrocarCenarioCorrente(cen2, t)\ \wedge\ \neg(cen == cen2))) \\ &\qquad \rightarrow CenarioCorrente(cen, t + 1)\end{aligned}$$

- Direção de objetos alinhados em x:

$$\begin{aligned}\forall o1, o2, t\ (&\exists x1, y1\ Posicao(x1, y1, o1, t) \wedge \\ &\exists x2, y2\ Posicao(x2, y2, o2, t) \wedge \\ &x1 = x2\ \wedge\ \neg(y1 = y2)) \\ &\qquad \rightarrow Direcao(o1, o2, 0, (y2 - y1)/Abs(y2 - y1), t)\end{aligned}$$

- Direção de objetos alinhados em y:

$$\begin{aligned}\forall o1, o2, t\ (&\exists x1, y1\ Posicao(x1, y1, o1, t) \wedge \\ &\exists x2, y2\ Posicao(x2, y2, o2, t) \wedge \\ &\neg(x1 = x2)\ \wedge\ y1 = y2) \\ &\qquad \rightarrow Direcao(o1, o2, (x2 - x1)/Abs(x2 - x1), 0, t)\end{aligned}$$

- Direção de objetos não alinhados:

$$\begin{aligned}\forall o1, o2, t\ (&\exists x1, y1\ Posicao(x1, y1, o1, t) \wedge \\ &\exists x2, y2\ Posicao(x2, y2, o2, t) \wedge \\ &\neg(x1 = x2)\ \wedge\ \neg(y1 = y2)) \\ &\quad \rightarrow Direcao(o1, o2, (x2 - x1)/Abs(x2 - x1), (y2 - y1)/Abs(y2 - y1), t)\end{aligned}$$

- Movimentação dos dragões:

$$\begin{aligned}\forall d, t\ &(Dragao(d) \wedge \\ &\neg(\exists t1, t2\ Mordendo(d, t1, t2)\ \wedge\ t >= t1\ \wedge\ t <= t2) \wedge \\ &\neg Morto(d, t)\ \wedge\ \neg Morto(personagem, t) \wedge \\ &(\exists cen\ (CenarioCorrente(cen, t)\ \wedge\ Encosta(d, cen, t))\ \vee\ d = yorgle)) \wedge \\ &\exists dx, dy\ Direcao(d, personagem, dx, dy, t) \\ &\quad \rightarrow Movimentar(dx, dy, d, t)\end{aligned}$$

- Efetivação da movimentação dos dragões:

$$\begin{aligned}\forall t\ &(\exists d\ Dragao(d)\ \wedge\ \exists drx, dry\ Posicao(drx, dry, d, t) \wedge \\ &\exists bc\ Boca(bc, d)\ \wedge\ \exists br\ Barriga(br, d) \wedge \\ &\exists dx, dy\ Movimentar(dx, dy, d, t) \wedge \\ &\quad \rightarrow Posicao(drx + dx, dry + dy, d, t + 1) \wedge \\ &\quad\ Posicao(drx + dx, dry + dy + 1, bc, t + 1) \wedge \\ &\quad\ Posicao(drx + dx + 1, dry + dy + 5, br, t + 1)\end{aligned}$$

- Definição da posição de dragões imóveis:

$$\begin{aligned}\forall t\ &(\exists d\ Dragao(d)\ \wedge\ \exists drx, dry\ Posicao(drx, dry, d, t) \wedge \\ &\exists bc\ Boca(bc, d)\ \wedge\ \exists br\ Barriga(br, d) \wedge \\ &\neg(\exists dx, dy\ Movimentar(dx, dy, d, t)) \wedge \\ &\quad \rightarrow Posicao(drx, dry, d, t + 1) \wedge \\ &\quad\ Posicao(drx, dry + 1, bc, t + 1) \wedge \\ &\quad\ Posicao(drx + 1, dry + 5, br, t + 1)\end{aligned}$$

- Ativação do estado de *morder* dos dragões:

$$\begin{aligned}\forall t\ &(\exists d\ Dragao(d)\ \wedge\ \neg Morto(d, t)\ \wedge\ \neg Morto(personagem, t) \wedge \\ &\neg(\exists t1, t2\ Mordendo(d, t1, t2)\ \wedge\ t >= t1\ \wedge\ t <= t2) \wedge \\ &Encosta(d, personagem, t)) \\ &\quad \rightarrow Morder(d, t)\end{aligned}$$

- Efetivação do estado de *morder* dos dragões:

$$\begin{aligned}\forall t\ &(\exists d\ Dragao(d)\ \wedge\ Morder(d, t) \wedge \\ &\exists drx, dry\ Posicao(drx, dry, personagem, t)) \\ &\quad \rightarrow Mordendo(d, t + 1, t + 11)\ \wedge\ Posicao(drx, dry - 1, d, t)\end{aligned}$$

- Definição do estado de *morto* dos dragões:

$$\begin{aligned}&\forall t\ (\exists d\ Dragao(d)\ \wedge\ \neg Morto(d,t)\ \wedge\\ &\neg(\exists t1,t2\ Mordendo(d,t1,t2)\ \wedge\ t >= t1\ \wedge\ t <= t2)\ \wedge\\ &\exists e\ Espada(e)\ \wedge\ Encosta(d,e,t))\\ &\rightarrow Morto(d,t)\end{aligned}$$

e

$$\forall t\ \exists o\ Morto(o,t) \rightarrow Morto(o,t+1)$$

- Definição do estado de *morto* do personagem:

$$\begin{aligned}&\forall t\ (\exists d\ Dragao(d)\ \wedge\ \exists t1,t2\ Mordendo(d,t1,t2)\ \wedge\ \neg Morto(d,t)\ \wedge\\ &\exists bc\ Boca(bc,d)\ \wedge\ Encosta(bc,personagem,t)\ \wedge\ t = t2\\ &\rightarrow Morto(personagem,t)\ \wedge\\ &Engolir(d,personagem,t)\end{aligned}$$

- Engolimento do personagem por um dragão:

$$\begin{aligned}&\forall t\ (\exists d\ Dragao(d)\ \wedge\ Engolir(d,personagem,t)\ \wedge\\ &\exists br\ Barriga(br,d)\ \wedge\ \exists brx,bry\ Posicao(brx,bry,br,t))\\ &\rightarrow Posicao(brx,bry,personagem,t+1)\end{aligned}$$

- Abertura de portões:

$$\begin{aligned}&\forall t\ (\exists ch,cor\ Item(ch)\ \wedge\ Chave(ch,cor)\ \wedge\\ &\exists po\ Portao(po,cor)\ \wedge\ Encosta(ch,po,t)\ \wedge\\ &\neg Aberto(po,t)\\ &\rightarrow Aberto(po,t+1)\end{aligned}$$

e

$$\forall t\ (\exists po,cor\ Portao(po,cor)\ \wedge\ Aberto(por,t))\ \rightarrow\ Aberto(por,t+1)$$

- Ativação de fim de jogo:

$$\begin{aligned}&\exists t\ (\exists ca\ Item(ca)\ \wedge\ Calice(ca)\ \wedge\\ &\exists cen\ CenarioCorrente(cen,t)\ \wedge\ cen = casteloAmarelo\ \wedge\\ &\neg FimJogo(t)\ \wedge\ ContemObjeto(cen,ca,t)\\ &\rightarrow FimJogo(t+1)\end{aligned}$$

e

$$\exists t\ FimJogo(t)\ \rightarrow\ FimJogo(t+1)$$

APÊNDICES B

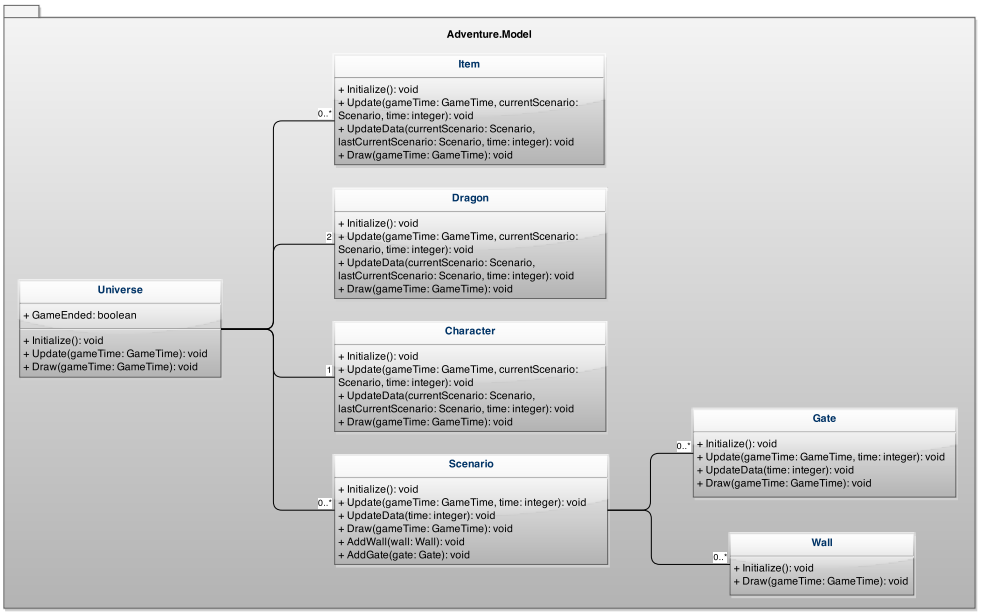


Figura 6.1: Diagrama de classes do pacote Adventure.Model da aplicação, no qual contém as classes de modelos.

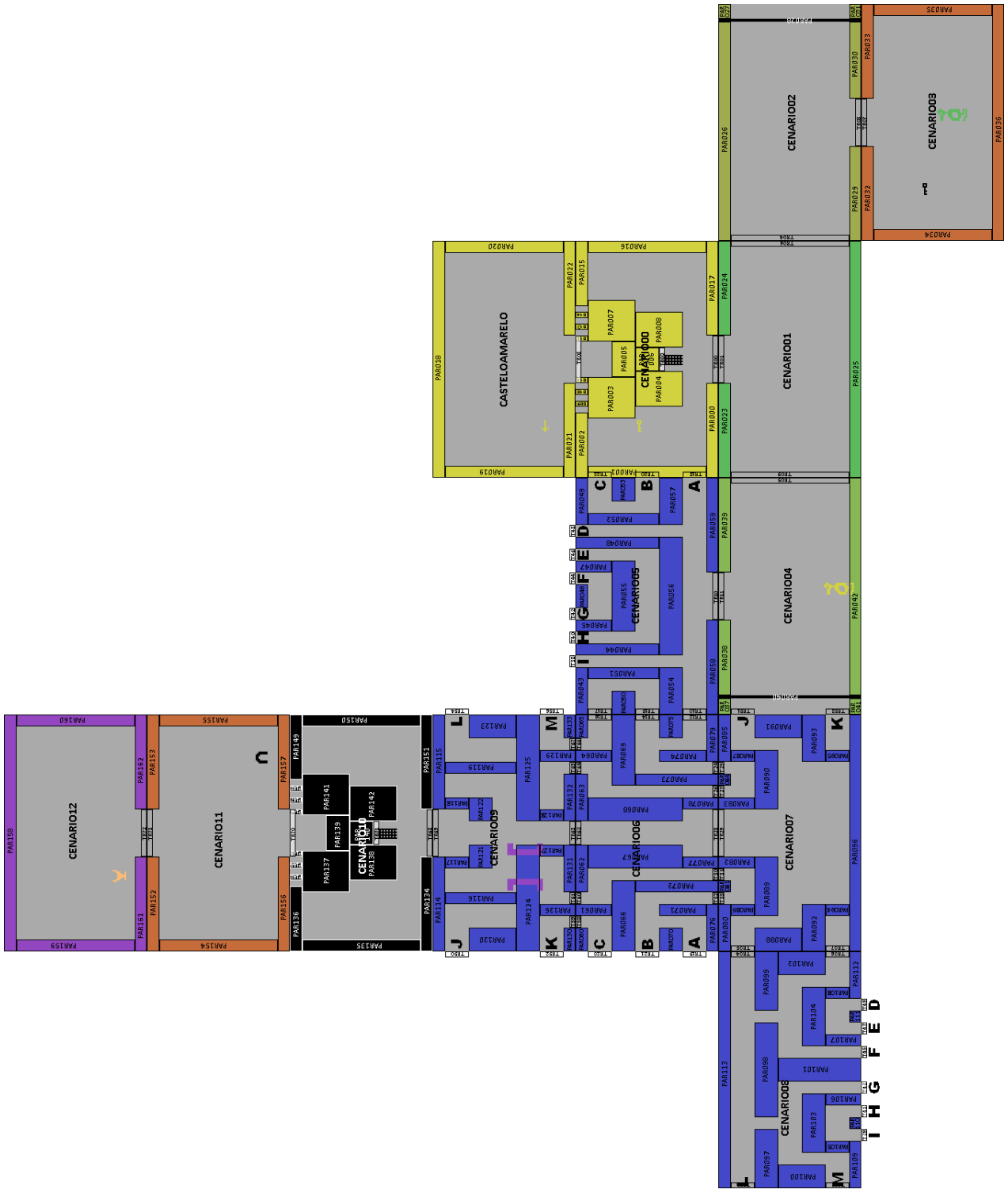


Figura 6.2: Mapeamento dos cenários, paredes, portões e transições do jogo.